IMPROVING CACHE PERFORMANCE WITH ADAPTIVE CACHE TOPOLOGIES AND DEFERRED COHERENCE MODELS

By

YONGJOON LEE

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1999

To
My parents and family

ACKNOWLEDGMENTS

Special thanks are due to my committee chairman, Dr. Jih-Kwon Peir, without whose unending patience, guidance, and expertise this effort would never have reached fruition. Thanks are also due to the rest of my committee, Dr. Timothy Davis, Dr. Kenneth O, Dr. Sanguthevar Rajasekaran, and Dr. Sartaj Sahni.

Many thanks are due to Windsor Hsu who helped to run a simulation program and Byung-Kwon Chung who set the multiprocessor simulation environment with decent efforts.

Also, the most appreciation is due to my wife, Chanwon, without whose patience, understanding and encouragement, I never would have embarked on, progressed though, and now completed the doctorate program and this effort.

TABLE OF CONTENTS

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

IMPROVING CACHE PERFORMANCE WITH ADAPTIVE CACHE TOPOLOGIES AND DEFERRED COHERENCE MODELS

By

Yongjoon Lee

August, 1999

Chairman: Jih-Kwon Peir
Major Department: Computer and Information Science and Engineering

Memory references exhibit locality and are therefore not uniformly distributed across the sets of a cache. This skew reduces the effectiveness of a cache because it results in the caching of a considerable number of less-recently used lines. In this dissertation, a technique that dynamically identifies these less-recently used lines and effectively utilizes the cache frames is described. These underutilized cache frames can be occupied by the more-recently used cache lines. Also, these frames can be used to further reduce the miss ratio through data prefetching. In the proposed design, the possible locations that a line can reside in is not predetermined. Instead, the cache is dynamically partitioned into groups. Because both the number of groups and each group associativity adapt to the dynamic reference pattern, this

design is called the adaptive group-associative cache. Performance evaluation shows the group-associative cache is able to achieve a hit ratio better than that of a 4-way set-associative cache. For some of the workloads, the hit ratio approaches that of a fully associative cache.

Private caches are a critical component to hide memory access latency in high performance multiprocessor systems. However, multiple processors may concurrently update a distinct portion of a cache line and cause unnecessary cache invalidations under traditional cache coherence protocols.

In this research, a deferred cache coherence model is described, which allows a cache line to be shared in multiple caches in the inconsistent state as long as the processors are guaranteed not to access any stale data. Multiple write requests to different portions of a cache line can be performed locally without invalidation. An efficient mechanism to reconcile multiple inconsistent copies of the modified line is described to satisfy the data dependence. Simulation results show that the proposed cache coherence model improves the performance of the parallel applications compared to conventional MESI and delayed coherence protocol.

In summary, a new adaptive cache topology which utilizes the cache frames and a new cache coherence model which minimizes the cache coherence activities are proposed. And the performance evaluation shows the proposed schemes improve the overall performance of the cache memory in both uniprocessor and multiprocessor systems.

CHAPTER 1
INTRODUCTION

1.1 Issues of Cache Memory

Programs exhibit both *temporal* locality, the tendency to reuse recently accessed data items, and *spatial* locality, the tendency to reference data items that are close to each other. These tendencies are called "*principle of locality.*" Because of this locality behavior of programs, a program tends to access a relatively small portion of its address space, normally referred as the *working set.*

For example, most programs which contain loops tend to access instructions repeatedly from the instruction cache, which shows high temporal locality. Instructions are normally accessed sequentially, that exhibit of high spatial locality. Also, accesses to elements of a data array as a record from the data cache in different iterations of a loop normally tend to have high degree of spatial locality.

A memory hierarchy which consists of multiple levels of memory with different access time and size is introduced to capture these locality behaviors. The faster memories are normally more expensive than slower memories. Therefore, it is advantageous to build memory systems as a hierarchical level, with the fast memory close to the processor and slower memory at lower levels. When the memory system is organized as a hierarchy, a level close to the processor is generally called a cache.

The performance of cache memory is critical to memory access time which affects the overall computer system performance.

The performance goal of adding a cache memory is to achieve an average memory access time approaching that of cache memory. The average memory access time is a measure of the time it takes to read a data item from the memory system. To achieve this goal, the majority of memory references should be satisfied by the cache, i.e., the cache miss ratio should be close to zero. This is possible because of the locality-of-reference property of memory reference streams, even though the size of cache is much smaller than the total size of memory address space of a program.

The average memory access time can be computed based on three parameters:

$$Average\ memory\ access\ time = Hit\ Time + Miss\ ratio \times Miss\ Penalty$$

1. *Hit time* : the time to access the cache memory, which includes the time to determine whether the access is a hit or miss.
2. *Miss ratio* : the fraction of memory accesses not found in the cache.
3. *Miss penalty* : the time to access the data from memory when the requested data is not present in the cache.

To reduce the average memory access time, the hit time, the miss ratio, and the miss penalty need to be reduced [21]. However, reducing cache hit time while reducing miss ratio is generally hard to achieve. To reduce miss ratio of the cache memory, cache memories are normally organized with complexity to present global locality of reference as accurately as possible. The global locality of reference defines that any data items in cache memory should be more-recently-referenced than the data item which is not in cache memory. This complexity normally makes the cache access

time to be increased because of complex hardware logics. The goal of a memory hierarchy is to reduce the overall memory access time, not just misses.

When a cache miss happens it can be categorized by one of compulsory miss, capacity miss, and conflict miss. The compulsory misses, capacity misses, and conflict misses are defined as follows by Hennessey and Paterson [22].

- Compulsory Misses – The first access to a block which is not in the cache, so the block must be brought into the cache, these are also called *cold start misses* or *first reference misses.*
- Capacity Misses – If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and data retrieved.
- Conflict Misses – If the block-placement strategy is set-associative or direct-mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called *collision misses.*

Restrictions on where a cache line is placed makes several conventional cache organizations. For each cache organization, the hit time and the miss ratio are different. If each line has only one place it can be placed in the cache, the cache is said to be *direct-mapped.* If a cache line can be placed anywhere in the cache, the cache is said to be *fully associative.* If a cache line can be placed in a restricted set of places in the cache, the cache is said to be *set-associative.* If there are n lines in a set, the cache organization is called *n-way set-associative.*

The direct-mapped cache topology can generally achieve faster cache hit time than set-associative cache or fully associative cache. This is due to the fact that the direct-mapped cache is simple and it searches only one location of the cache to find whether a reference is a hit or not. An advantage of a set-associative cache over the

direct-mapped cache is that it is much less likely to encounter collision situations. If two frequently referenced objects happen to collide in a direct-mapped cache, then those two referenced objects replace each other. Those two objects can be retained in a 2-way set-associative cache design. However, the hit time to access the set-associative is slower than that of the direct-mapped cache since the set-associative cache need to compare more than one address tags simultaneously for each reference.

The fully associative cache design has an advantage over the direct-mapped and set-associative cache. In fully associative cache, a cache block can be placed anywhere in the cache. Thus, the fully associative cache can avoid the conflict misses unlike direct-mapped cache or set-associative cache. However, in order to implement the fully associative cache, the cache must access all address tags to find whether it is a hit or miss for each reference. Also, it should maintain proper order of entries in the cache to replace the least-recently used cache line upon a miss. These searches and maintaining the order of cache contents need much more time to access the cache than other cache organizations.

The global Least-Recently-Used(LRU) replacement policy, which is used in the fully associative cache to replace the least-recently used cache line from cache memory upon a miss, has generally lower cache miss ratios than any other cache replacement topologies. The other schemes, such as direct-mapped cache or set-associative cache are an approximation of the global LRU replacement. The set-associative cache is partitioned into distinct sets of lines and each set contains a

small fixed number of lines. Hence, the set-associative cache has several fixed places to place a new line when a cache miss happens. The LRU replacement scheme is performed within each set.

Experiments showed that the direct-mapped cache and set-associative cache can not hold the more-recently used cache lines like global LRU replacement policy. The rationale is that the more-recently used cache lines are not distributed evenly throughout the entire cache sets. As a result, many locations of the direct-mapped cache and set-associative cache include the contents which do not exist in the fully associative cache. These cache contents, in general, have very little chance to be referenced before replacement.

In this dissertation research, a new cache organization, which reduces the miss ratio of cache memory without increasing cache hit time, is introduced and proposed. This new cache organization is called *Group-Associative Cache*. The group-associative cache maintains the fast access of the direct-mapped cache while reducing its miss ratio. Like the direct-mapped cache and the set-associative cache, the group-associative cache is also an approximation of the fully associative cache. However, the group-associative cache can approximate the global LRU replacement policy more accurately without maintaining precise reference sequences among all the cache lines.

Furthermore, another way to improve cache performance is *Data Prefetching* which reduces the miss penalty by fetching data from a predicted address before

actual using the data from the processor. In this dissertation research, data prefetching to the new cache topology, group-associative cache, is introduced and proposed. The fact that patterns of program execution are largely sequential provides the opportunity to predict the addresses which are likely to be accessed in near future. Data prefetching can hide memory latency because the data prefetching can bring the data from a lower memory hierarchy before a miss happens. And the transfer may be overlapped with instruction execution. However, if the location to place the prefetched data is the place which holds the more-recently used cache line, then the data prefetching might reduce the performance of the cache memory by replacing such a recently used line. And this replacement by the prefetched data may replace the prefetched data without any reference. Since the cache is a very valuable resource in the computer system the place to put the prefetched data may affect the performance of cache memory.

The prefetched cache line may *pollute* the cache memory if the prefetched cache line does not have a chance to be referenced before it is replaced from the cache memory. This pollution may increase the cache miss ratios, even though the prefetched line might be referenced in the near future. If the cache frame to place the prefetched cache line is currently occupied by a more-recently used cache line, then there is the possibility to collide with the current cache line and the prefetched cache line. This collision replaces each other and reduces the performance of the cache. This collision pollutes the cache memory. If the prefetched cache line is not referenced, then the bandwidth used to bring the prefetched cache line is wasted.

Therefore, the place to put prefetched cache line is one of the important performance factors for data prefetching.

Processors can be interconnected to form a multiprocessor system which is composed of the processors and connection mechanism. When the multiprocessor does not use local memories, the cache does a vital role in reducing the contention for the shared system bus. Without cache memories, typically the instructions and data might be accessed by the processor using the system bus all the time. When each processor has independent cache memory, the same cache line might exist on two or more processor's cache. If these coexisted cache lines are different (inconsistent) version of the same cache lines at the same time, this is called *cache coherency problem.*

The *cache coherence protocol* is a protocol of how to solve the cache coherency problem in the multiprocessor systems. When a cache line is shared, there is true and false sharing. True sharing is that a processor accesses a value that is written by a different processor. False sharing is when a line is modified since the last time it was in the processor's cache, but the processor does not use any of the newly written values [61]. The false sharing behavior depends on how the memory reference from multiple processors are interleaved. When a cache line is shared by several processors, each processor might use a different portion of the cache line. For false sharing cache lines, the invalidations for every write to the shared cache line are not necessary. This invalidation causes false sharing miss to other processors which share the cache line at the same time.

The Single-Program-Multiple-Data(SPMD) programming model provides synchronization primitives which are used to satisfy the dependency and consistency of the cache lines. Hence, within each section between synchronizations, the shared cache line can be partially modified to a different portion of the cache line. And inconsistent cache lines are needed to be consistent when the line is accessed after a synchronization primitive. This observation points out that the partially modified cache line can exist between synchronization primitives. Also, for these partial modified cache lines, an efficient way to make consistent state from inconsistent cache lines is proposed.

In this dissertation research, a new cache coherence model, which is called *deferred cache coherence model* is introduced and proposed under Symmetric Multi-Processor systems(SMP) environment. The new cache coherence model is a hardware based approach to improve the performance of parallel applications.

The deferred cache coherence model adds new states to the existing cache coherence protocol to defer the write-invalidation and to allow simultaneous reads/writes to the different portion of the same line in multiple caches. An efficient technique is proposed to reconcile cache lines which are written by several processors. Using this new cache line states and the reconciliation technique, the deferred cache coherence protocol allow multiple writes to different portion of the cache line which is shared among several processors.

When there is a false sharing cache line, the cache line might be invalidated for each write without actual data sharing. Also, a line might be transfered between

caches when there is a read after write(s). The deferred cache coherence protocol allows reads/writes by several processors for the false shared cache line until the laziest time to be consistent. Thus, the deferred cache coherence protocol prevent unnecessary invalidations and transfers under SPMD programming model.

1.2 Performance Evaluation Methods

In order to evaluate the performance of the proposed cache schemes, it is needed to simulate the standard performance benchmark programs using tracing facilities.

1.2.1 Tracing Facilities

The tracing facilities which are used in this dissertation to simulate the benchmark programs for the cache studies are two kinds of simulation tools. For the uniprocessor model, a tracing tool *Shade* [56] is used to simulate the performance of cache memories. The Shade runs on Sun's SPARC/SOLARIS environment. As an application program runs, an instruction trace record for the executed instruction is transfered to the target system simulator. Shade controls the application program execution.

Figure 1.1 shows the shade execution configuration. The target system simulator receives a trace record from memory reference generator, then the target system simulator simulates the instruction of the received trace record. Shade, the

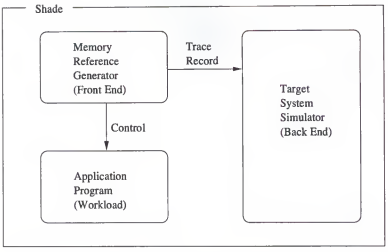


Figure 1.1. A trace-driven simulator Shade

trace-driven simulator, can only trace the uniprocessor programming model and the target system simulator does not signal to memory reference generator. The target system simulator only receives trace record and it simulates the record. The memory reference generator is also called *tracer* or *front end*. And the target system simulator is also called *analyzer* or *back end*.

In the study of multiprocessor memory hierarchy, a program-driven simulator was used to simulate the performance of proposed cache scheme. *Mint* ("Mips interpreter") is a multiprocessor program-driven simulator and it controls the scheduling of processes so that the interleaving of memory reference is the same as it would be on the simulated machine [59, 60]. The Mint is partitioned into two main parts: a memory reference generator ("front end"), and a target system simulator ("back end"). The memory reference generator models the execution of an application program on some number of processors. When the front end performs an operation like a memory read, it sends an event to the back end. The back end analyzes the target

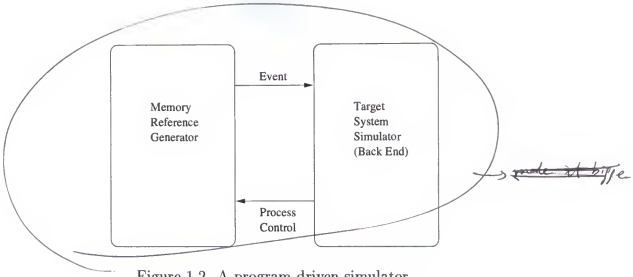


Figure 1.2. A program-driven simulator

system using the received event. When the back end completes the simulation on the event, it signals to the front end, so that some processes can continue. However, if an event can't be executed from back end, the event must be rescheduled to send again from front end to back end. To do this, the back end signals to front end to control the process of the front end. Figure 1.2 shows a general configuration of program-driven simulation like Mint.

1.2.2 Workloads

Benchmark programs are intended to provide a measure to compare performance. Typically, a standardized benchmark is used, so performance can be compared with other models. The Standard Performance Evaluation Corporation(SPEC)-95 [55] is one of the standardized benchmark programs widely used to simulate a uniprocessor system model. The SPEC95 products are composed of two suites of benchmarks with 18 applications. Among these 18 applications there are 8 integer

intensive programs (*Compress, Gcc, Go, Ijpeg, Li, M88ksim, Perl* and *Vortex*), and 10 floating-point intensive programs (*Applu, Apsi, Fpppp, Hydro2d, Mgrid, Su2cor, Swim, Tomcatv, Turb3d* and *Wave5*). Short explanations of the SPEC95 benchmark programs follows. A commercial workload is also used for simulations.

1. SPEC95 integer benchmark programs (SPECint95)
   - 099.Go : Artificial intelligence, plays the game of "Go"
   - 124.M88ksim : Moto 88K Chip Simulator, runs test program
   - 126.Gcc : New Version of GCC, builds SPARC code
   - 129.Compress : Compresses and decompresses file in memory
   - 130.Li : LISP interpreter
   - 132.Ijpeg : Graphic compression and decompression
   - 134.Perl : Manipulates strings (anagrams) and prime numbers in Perl
   - 147.Vortex : A database program
2. SPEC95 floating point benchmark programs (SPECfp95)
   - 101.Tomcatv : A mesh-generation program
   - 102.Swim : Shallow water model with 1024 X 1024 grid
   - 103.Su2cor : Quantum physics, Monte Calos simulation
   - 104.Hydro2d : Astrophysics, Hydro-dynamical Navier Stokes equations
   - 107.Mgrid : Multi-grid solver in 3D potential field
   - 110.Applu : Parabolic/elliptic partial differential equations
   - 125.Turb3d : Simulation isotropic, homogeneous turbulence in a cube
   - 141.Apsi : Solves problems regarding temperature, wind, velocity, and distribution of pollutants
   - 145.Fpppp : Quantum chemistry
   - 146.Wave5 : Plasma physics, Electro magnetic particle simulation
3. Database workload
   - With the help of Windsor Hsu at IBM Almaden Research Center, a simulation ran a workload similar to Transaction Processing Performance Council's Benchmark C.

For the multiprocessor simulation, the Stanford ParalleL Application for SHared (SPLASH)-2 parallel applications are widely used [61]. The SPLASH-2 parallel applications are to facilitate distributed-address-space multiprocessors. The SPLASH-2 benchmark suite consists of mixture of complete applications and computational kernels, which present computations in scientific, engineering, and graphic computing. Short explanations of the SPLASH-2 parallel benchmark programs follows.

- Barnes : The Barnes application simulates the interaction of a system of bodies in three dimensions over a number of time-steps.
- Cholesky : The blocked sparse Cholesky factorization kernel factors a sparse matrix into the product of a lower triangular matrix and its transpose.
- FFT : This is a Fast Fourier Transform algorithm. The data set consists of the n complex data points to be transformed.
- FMM : The FMM application simulates a system of bodies over a number of time-steps in two dimensions.
- LU : The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix.
- Ocean : The Ocean application simulates large-scale ocean movements based on eddy and boundary currents.
- Radix : The integer radix sort algorithm
- Raytrace : This application renders a three-dimensional scene using ray tracing.
- Volrend : This application renders a three-dimensional volume using a ray casting technique.
- Water : This application evaluates forces and potentials that occur over time in a system of water molecules.

1.2.3 Machine Models

For the uniprocessor performance simulation, the separate and identical instruction and data L_1 caches are simulated. The size of the L_1 cache ranges from 8 to 64 KB with set-associativities of 1, 2, and 4. Fully-associative caches are also considered. Those on-chip L_1 caches are backed up by a 512 KB 4-way set-associative unified level 2 (L_2) cache. The line sizes of the L_1 and L_2 caches are 32 bytes and the LRU replacement policy is used in all the cases. Inclusion property is enforced between L_2 and L_1 data caches.

For the group-associative cache, the different configurations of group-associative cache are simulated to see how a configuration of the group-associative cache affects the performance. In order to compare with other enhanced cache topologies, the victim cache [29] and the column-associative cache [3] are modeled. For the victim cache, extra space is used to keep the lines which are victim lines. For the column-associative cache, the secondary location is determined by flipping the highest order index bit, the rehash bit is included with each entry of the tag array to guide the search and replacement.

For multiprocessor performance simulation, the split-transaction snooping cache based on the MESI and the deferred coherence models are modeled. The snooping bus consists of a *command/address bus* (or simply *command bus*) and a *data bus*. The width of the data bus is considered under current technology. The spilt-transaction snooping bus sends requests to the other processors, after receiving

acknowledgment the requester finishes its bus transaction. The processors, which receive requests from snooping bus, do the cache lookup and update.

In view of current technology, it is assumed that the processor cycle is four times faster than that of the bus cycle and a bus request can be issued in every three bus cycles. It is assumed that each instruction takes one cycle to execute using a perfect branch prediction when the instruction is found in L_1 instruction cache. The load/store instruction also takes a single cycle if both the instruction and the data are found in the L_1 cache. When the instruction or data is not found in L_1 cache but found in unified L_2 cache, four cycles are charged to bring the instruction or data to L_1 cache.

For the command bus request, once a bus request is active, it requires two cycles for bus arbitration. The command and address is issued right after the bus is granted. It then takes two cycles for each processor to look-up and update the cache directory for the snooping bus request. Under these assumptions of processor and bus transaction models, some cycle-by-cycle program-driven simulation models are implemented.

1.2.4 Overview of the Dissertation

In presenting the material, this dissertation is composed of five chapters, each concentrating its contents on a specific area which affects the overall performance

of the cache memories. The first chapter offers an overview of the issues of cache memories and the methods of evaluation the proposed designs.

Chapter two offers the group-associative cache design, which can capture the dynamic memory reference behaviors while maintaining the fast access time of direct-mapped cache design. The group-associative cache design is a new cache design and it utilize the cache frames effectively, so the proposed cache design improves the performance of the cache memory.

Chapter three presents the data prefetching, which is a way of reducing cache miss ratios. It emphasizes data prefetching using group-associative cache design improving the hit ratio remarkably without extra space for data prefetching.

Chapter four is a study of the cache coherence model. The proposed cache coherence model, the deferred cache coherence model, improves the execution of the parallel programs. The deferred cache coherence model reduces the requests to the system bus too. Finally, the last chapter draws conclusions of the dissertation.

CHAPTER 2
CAPTURING DYNAMIC MEMORY REFERENCE BEHAVIOR WITH ADAPTIVE CACHE TOPOLOGY

2.1 Introduction

The least-recently used (LRU) replacement policy, which replaces the least-recently used cache line when the requested line does not exist in the cache, works very well for memory hierarchy caching schemes because of the locality of reference. However, for the processor caches where access time and hardware complexity are major design issues, a global LRU replacement policy across the entire cache is impractical. Instead, the cache is typically organized into sets of cache lines within which the LRU replacement policy is used. In this dissertation research, it is observed that such an organization is a poor approximation of the global LRU replacement policy because the more-recently-referenced cache lines are not evenly distributed across all the cache sets. This non-uniform distribution results in the caching of a significant number of less-recently used lines which are less likely to be re-referenced before replacement. Such an effect is especially pronounced for the direct-mapped cache which, because of its fast access time, is often the cache topology of choice for first level cache [23]. Simulation results using SPEC95 (The Standard Performance Evaluation Corporation) benchmark suites [55] have shown

that on average about 35 % of direct-mapped cache frames are containing less-recently used lines during execution. Such lines have less than a 0.7 % chance to be reused before they are replaced. The ensuing impact on cache hit ratio is considerable, especially for direct-mapped caches.

In this chapter, an *adaptive group-associative cache* is described that more accurately approximates the global LRU replacement policy by using a history table to keep track of the cache lines that have been referenced recently. When a cache miss occurs and the line being replaced has been referenced recently, it is moved into an alternate location within the cache. The alternate location is selected from among those that have not been accessed in the recent past. A small directory is used to keep track of the more-recently used lines that have been displaced. In this design, the possible locations that a line can reside in is not predetermined as is the case in a set-associative cache. Instead, the cache is dynamically partitioned into groups of cache lines with the same index bits. Because the total number of groups and the individual group associativity adapt to the reference pattern, this design of cache is called the adaptive group-associative cache.

2.2 Statement of Problem

Even though the global LRU replacement policy works extremely well from the miss ratio point of view, which is one of major factors of cache performance. The direct-mapped caches tend to be adopted by most of high performance computers

since the direct-mapped caches are faster, simpler, and easier to design than set-associative cache or fully associative cache. Also, the direct-mapped cache requires less space to build. However, the direct-mapped cache tends to have a lower hit ratio than fully associative cache and set-associative cache with LRU replacement policy due to conflict misses [23]. In direct-mapped cache, each line has only one location it can be placed in the cache, so the conflicts happen frequently and the conflict misses are the most misses among all misses in direct-mapped cache. Also, it is observed that in direct-mapped cache about 35 % of blocks are not more-recently used blocks compared to the fully associative cache, which can keep all more-recently used lines in its cache from the global point of references. Those less-recently used blocks are hard to be re-referenced before replacement because of the locality of reference. In this proposal, those blocks which contain less-recently used block are called *Empty Blocks* or *Holes*.

2.2.1 Cache Miss Behaviors

When a cache miss occurs, the miss can be categorized as a compulsory, capacity, or conflict miss. The miss ratio and the relative percentage of these misses are different for each cache design. When the set-associativity of cache is increased, the conflict misses tend to be reduced without increasing the size of cache. However, increasing the set-associativity of the cache tends to increase the hit time of cache, which is one of the major factors of cache performance. Even though the

fully associative cache reduces the conflict misses completely, the fully associative cache becomes very expensive and increases the hit time of the cache. These reasons make fully associative cache hard to be built [53].

In order to understand the compulsory, capacity, and conflict misses with different cache topology and cache size, various cache topologies are simulated using SPEC95 benchmark programs. These are direct-mapped cache, 2-way set-associative cache, 4-way set-associative cache, and fully associative cache. The LRU replacement policy is adopted to compute capacity misses. The Tables in the Appendix show that the compulsory, capacity and conflict misses of L1 data and instruction cache for the SPEC95 workloads. The statistics by simulation were collected from 2 billion instructions after skip 2 billion instructions. Caches were warmed-up by 25 million instructions.

Table 2.1 and 2.2 show the relative percentage of the compulsory, capacity, and conflict misses. The cache size is 16 KB data cache. From the Tables, it is observed that the data reference of the most workloads, Apsi, Fpppp, Gcc, Go, Ijpeg, Li, M88ksim, Swim, Turb3d, Vortex, and Wave5, have relatively high percentage of conflict misses. For these workloads, more than 50 % of total misses are defined as conflict misses when the direct-mapped cache is used. The set-associativity cache reduce the conflict miss percentage however, the two-way set-associative cache has a relatively higher percentage of conflict misses. Some workloads, Applu, Compress, Hydro2d, Mgrid, and Su2cor, have a lower percentage of conflict misses. For these workloads the capacity misses are the dominant miss type among all the misses.

Table 2.1. The relative percentage of compulsory, capacity, and conflict misses of SPECint95, 16KB data cache

BENCHMARK	ASSOCI-ATIVITY	COMPULSORY* MISS (%)	CAPACITY MISS (%)	CONFLICT MISS (%)
Compress	D.M.	1.52	84.2	14.3
	2-way	1.74	96.2	2.10
	4-way	1.75	97.1	1.15
Gcc	D.M.	0.26	46.7	53.1
	2-way	0.40	72.1	27.5
	4-way	0.48	87.2	12.3
Go	D.M.	0.01	42.6	57.4
	2-way	0.02	68.5	31.5
	4-way	0.02	87.3	12.6
Ijpeg	D.M.	1.59	45.5	52.9
	2-way	3.12	89.3	7.57
	4-way	3.16	90.6	6.72
Li	D.M.	0.03	52.8	47.2
	2-way	0.05	86.9	13.1
	4-way	0.06	97.4	2.58
M88ksim	D.M.	1.29	3.9	94.8
	2-way	4.4	13.2	82.5
	4-way	9.67	29.2	61.1
Vortex	D.M.	2.0	34.1	63.9
	2-way	3.31	56.2	40.5
	4-way	4.60	77.8	17.6

* Note due to the skip the initial phase and warm-up caches as well as simulation 2 billion instructions the Compulsory misses may not be very accurate.

Table 2.2. The relative percentage of compulsory, capacity, and conflict misses of SPECfp95, 16KB data cache

BENCHMARK	ASSOCI-ATIVITY	COMPULSORY* MISS (%)	CAPACITY MISS (%)	CONFLICT MISS (%)
Applu	D.M.	1.76	84.3	14.0
	2-way	2.03	97.2	0.74
	4-way	2.05	98.0	0
Apsi	D.M.	0.07	28.7	71.3
	2-way	0.12	49.3	50.6
	4-way	0.13	55.6	44.3
Fpppp	D.M.	0	12.8	87.2
	2-way	0	47.8	52.2
	4-way	0	99.7	0.23
Hydro2d	D.M.	0.25	92.8	6.97
	2-way	0.26	97.6	2.14
	4-way	0.27	99.7	0
Mgrid	D.M.	0.53	80.8	18.7
	2-way	0.66	99.3	0
	4-way	0.65	99.6	0
Su2cor	D.M.	0.33	90.1	9.59
	2-way	0.36	99.0	0.6
	4-way	0.36	98.8	0.89
Swim	D.M.	0.20	32.5	67.3
	2-way	0.18	29.4	70.4
	4-way	0.17	27.9	72.0
Tomcatv	D.M.	0.36	78.8	20.9
	2-way	0.34	72.6	27.1
	4-way	0.37	80.8	18.9
Turb3d	D.M.	2.30	31.2	66.5
	2-way	2.70	36.9	60.4
	4-way	3.81	51.8	44.4
Wave5	D.M.	0.64	49.5	49.9
	2-way	0.64	49.3	50.1
	4-way	0.64	49.5	49.9

* Note due to the skip the initial phase and warm-up caches as well as simulation 2 billion instructions the Compulsory misses may not be very accurate.

These workloads have a much bigger working set size, so the working set does not fit in the cache.

For the instruction stream of the SPECfp95 workloads, the miss ratio to the instruction cache is less than 1 % except for the workload Fpppp. Thus the instruction stream of the SPECfp95 workloads does not affect the performance of the workload execution. For the workload Fpppp, the instruction stream shows that the capacity miss is the highest percentage of misses.

For the instruction stream reference of the workload SPECint95, the instruction cache results show that the workloads, Gcc, Go, Li, M88ksim, Perl, and Vortex, have relatively high conflict misses. Two workloads, Compress, and Ijpeg, have very little instruction cache misses, so the instruction stream does not affect the execution of the programs. The negative numbers on "Conflict misses" field show that the miss ratio of the fully associative cache is larger than that of the miss ratio of corresponding cache configuration. For these cases, the "Capacity misses" is the major miss and the conflict miss is very small compared to "Capacity misses". The detail numbers can be found in Appendix A and B.

2.2.2 Underutilized Cache Frames

The performance of a cache is determined both by the fraction of memory requests it can satisfy and the speed at which it can satisfy them. The simple direct-mapped cache provides a fast access time but tends to have a low hit ratio

due to conflict misses [23]. In a direct-mapped cache, a line can only be located at a fixed location within the cache. This restrictive line placement means that lines that have been assigned the same cache location have to replace one another, even though those lines have been referenced very recently. In other words, the direct-mapped cache is unable to always retain the set of more-recently used lines. This inability to retain all the more-recently used lines can be quantified by mapping the contents of a fully associative LRU-replacement cache to a direct-mapped cache. As shown in Figure 2.1, the proper index bits from each line in the fully associative cache determine the corresponding location in the direct-mapped cache. In a typical case, the more-recently used lines are not evenly distributed across all the sets in the direct-mapped cache or set-associativity cache. For instance, several lines (*a, b, c,* and *d*) cannot fit into the direct-mapped cache. As a result, the direct-mapped cache contains a number of empty blocks or *holes*. In real operation, these holes will be filled with other less-recently used lines which have less chance of being re-referenced before replacement.

Figure 2.2 thru Figure 2.4 plot the average percentage of holes that exist in various cache configurations. The workloads of this experiment are the Gcc, Vortex, and Turb3d of SPEC95 benchmark programs [55]. The Gcc and Vortex are workloads of SPECint95. The workload Gcc is a compilation application and the workload Vortex is a database application. The workload Turb3d belongs to SPECfp95, and it is a simulation application, which simulates turbulence in a cubic area. The results show that a very significant amount of holes exist in both the instruction and

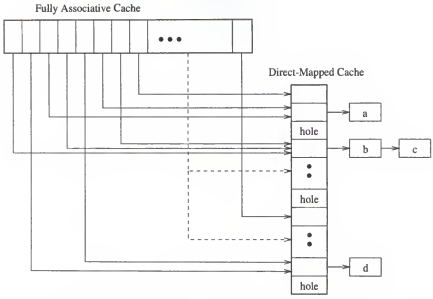


Figure 2.1. Mapping a fully associative cache to a direct-mapped cache.

data caches. For the direct-mapped data caches, between 29.6 % and 40.3 % of the cache are holes. The corresponding ranges for the 2-way and 4-way set associative caches are about 14.2 – 26 % and 8.6 – 20 % respectively. Compared with the data caches, 31.6 – 35.6 % of the direct-mapped instruction caches are holes. The corresponding numbers for the 2-way and 4-way set associative instruction caches are about 23.3 – 25.9 % and 15.4 – 17.8 % respectively. The percentage of holes in the data caches increase a little with larger cache sizes except the workload Gcc, while the percentage of holes in the instruction caches seems to be insensitive to cache size. For the workload Turb3d, since the instruction cache miss ratio is almost zero, the percentage of the holes in the instruction cache are not plotted. Even though the percentage of holes is more than 30 % of the direct-mapped cache, the hit percentage to these holes is mostly less than 0.7 %. Table 2.3 shows the percentage of

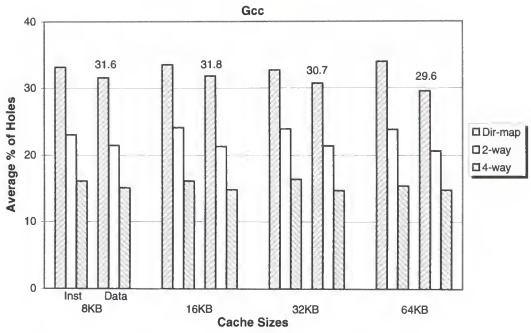


Figure 2.2. Percentage of holes for Gcc of SPECint95 benchmarks.

hit to the holes for Gcc, Vortex and Turb3d for the first level direct-mapped data cache.

Figure 2.5 and Figure 2.6 show the percentage of holes for the rest of the workloads except Gcc, Vortex, and Turb3d. For Figure 2.5 and Figure 2.6 the cache size is 16 KB data cache. From the figures, when the cache scheme is direct-mapped cache, all the workloads have a large percentage of the holes. From this experiment, it is observed that 25 % to 60 % of cache frames are defined as holes. For the two-way set-associative cache, about 25.2 % of cache frames are defined as holes, and this number is about 18.1 % for the four-way set-associative cache average.

In SPEC95 workloads, the Ijpeg and Applu have a lower percentage of the holes for all the data cache sizes than that of the other workloads. For the workloads in SPECfp95, Su2cor has the highest percentage of the holes especially when the

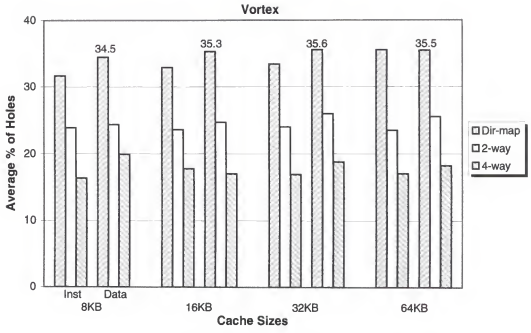


Figure 2.3. Percentage of holes for Vortex of SPECint95 benchmarks.

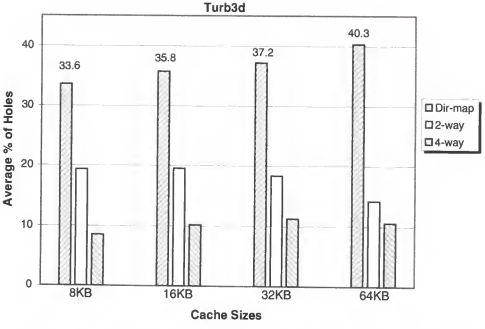


Figure 2.4. Percentage of holes for Turb3d of SPECfp95 benchmarks.

Table 2.3. Hit ratio to the holes (L_1 data cache, direct-mapped)

WORKLOAD	CACHE SIZE	% OF HOLES	HIT RATIO
Gcc	8KB	31.6 %	0.65 %
	16KB	31.8 %	0.39 %
	32KB	30.7 %	0.12 %
	64KB	29.5 %	0.16 %
Vortex	8KB	34.5 %	0.19 %
	16KB	35.3 %	0.07 %
	32KB	35.6 %	0.09 %
	64KB	35.5 %	0.03 %
Turb3d	8KB	33.6 %	0.044 %
	16KB	35.8 %	0.154 %
	32KB	37.2 %	0.483 %
	64KB	40.3 %	0.005 %

direct-mapped cache configuration is used. For the workload, Perl, when the cache is 16 KB fully associative cache the miss ratio is zero, so the percentage of the holes can not be calculated.

As discussed above, the existence of such holes have a considerable impact on the cache miss ratio. In addition, it limits the performance impact of hit ratio improvement techniques such as the column-associative cache and the victim cache. The column-associative cache is a direct-mapped cache in which each set has an alternate backup set that is accessed through a secondary hash function [3]. When a memory request misses in the primary set, the column associative cache is accessed again with the secondary hash function. A *rehash* bit is included with each tag entry to indicate whether the line is to be accessed through the secondary hash function. In essence, the column-associative cache is effectively a 2-way set associative cache

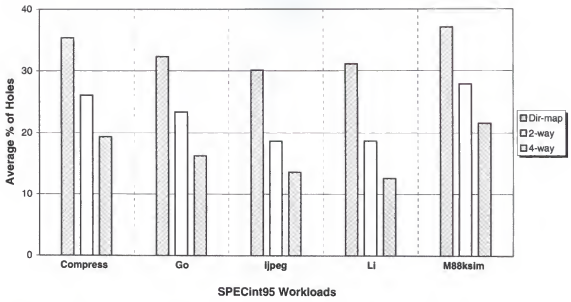


Figure 2.5. Percentage of holes for SPECint95 benchmarks (16KB data cache).

which still contains a significant number of holes. The victim cache is a separate fully associative buffer that holds the recent victims of replacement, i.e., the lines that have been recently evicted from the direct-mapped cache [29]. In this approach, the less-recently used lines that are evicted from the direct-mapped cache will fill the victim cache, thereby polluting the victim cache and reducing its effectiveness. Moreover, a large number of holes remain in the direct-mapped cache.

2.3 Adaptive Group-Associative Caches

The basic idea behind the adaptive group-associative cache is to maintain the fast access time of the direct-mapped cache while improving its hit ratio by using the existing *holes* to store the lines that have been recently referenced. There are thus three parts to this scheme. The first is to dynamically identify the holes. The

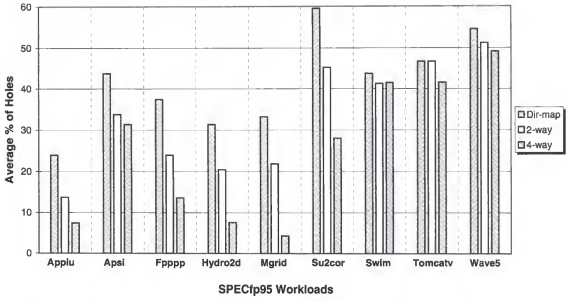


Figure 2.6. Percentage of holes for SPECfp95 benchmarks (16KB data cache).

second is to determine whether a line should be evicted from the cache or placed into a hole upon replacement. The third is to locate the *out-of-position* lines, i.e., the lines that have been displaced from their direct-mapped locations, into holes. If the majority of holes are correctly identified and filled with more-recently used lines, the group-associative cache can achieve a hit ratio approaching that of a fully associative cache.

A straightforward approach to implementing the group-associative cache is to maintain two small directories. One directory, the *Set-reference History Table (SHT)*, tracks the sets that have been referenced recently. The other directory, the *Out-of-position Directory* or *OUT directory* for short, records the tags and locations of the lines that have been recently displaced from their direct-mapped positions. When a miss occurs in a set that is tracked by the SHT, the line to be replaced

is not evicted from the cache but is instead moved to another location within the cache. The rationale for this is that the replaced line must have been referenced recently for its set to be recorded in the SHT. In order for this displaced line to be located, its tag and new location or set-id is entered into the OUT directory. When a line is neither recorded in the SHT nor the OUT directory, it is said to be *disposable*. Disposable lines are the candidates for eviction when a miss occurs.

In a group-associative cache, the lines with identical direct-mapped index bits belong to a *congruence group*. The total number of groups and the number of lines within each group adapt dynamically to the reference pattern. This is in contrast to conventional cache organizations where the number of sets and the set-associativity are fixed. For instance, if the reference pattern is such that different lines within the same set are continually accessed, all the out-of-position lines may belong to the same congruence group. When there is more than one line in a congruence group, all but one must be located out of the direct-mapped location. The line in the direct-mapped position is located through the cache tag array, as is the case in the column-associative cache, a fast access time can be achieved in this case. The out-of-position lines are located through the OUT directory which is searched in parallel with the cache tag array. Besides determining hit or miss, the OUT directory also provides the location of the out-of-position lines. On a hit to an out-of-position line, the data array is accessed again with the correct set-id obtained from the OUT directory.

The size and topology of the SHT and the OUT directory are flexible and can be designed based on the overall performance as well as the area cost. The detailed operations of the adaptive group-associative cache and its performance impact will be discussed next.

2.3.1 An Example Design

Figure 2.7 shows the block diagram of a straightforward implementation of the adaptive group-associative cache. The SHT tracks the sets that have been referenced recently. Since each set contains a single line in a direct-mapped cache, the SHT remembers the set of more-recently used lines that are located in their direct-mapped positions. The OUT directory contains the address tag and the set-id of the lines that have been recently displaced from their direct-mapped locations. In other words, the SHT and the OUT directory together define the set of lines that are not disposable, i.e., the set of lines that have been referenced recently and therefore should not be evicted. By design, a line cannot be recorded in both the SHT and the OUT directory at the same time. To simplify cache management, a *disposable* or *d* bit is attached to each entry in the cache tag array to indicate whether the line should be evicted when it is replaced.

For the group-associative cache, there are three kinds of situations below. In these situations, the group-associative cache maintains SHT and OUT directory

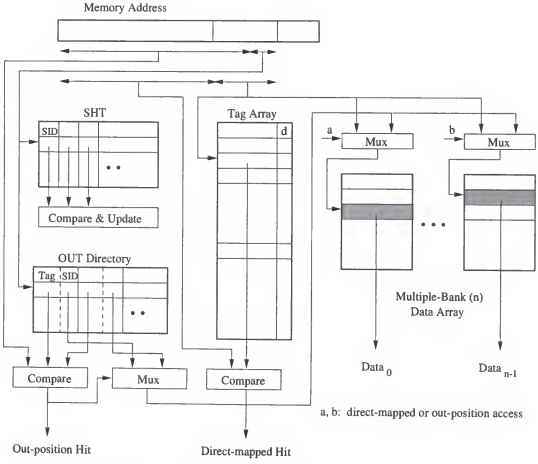


Figure 2.7. An Adaptive Group-Associative Cache.

correctly, and the most-recently used (MRU) line of each congruence group will be in direct-mapped location.

Hit in Direct-Mapped Location. When a memory request occurs, the cache tag array and the OUT directory are searched in parallel. If there is a match in the cache tag array, the data is accessed as in a regular direct-mapped cache. The SHT is always updated after a reference to reflect the most-recently used line.

Hit in Out-Of-Position Lines. If the line is found through the OUT directory, the data is accessed using the set-id fetched from the OUT directory. In this case, the requested line is swapped with the line located in the direct-mapped location so as to increase the hits to the direct-mapped position. The OUT directory is updated to record the new out-of-position line. Note that unlike the fully associative cache in which all the index bits are part of the address tag, none of the index bits is included in the tag array in the group-associative cache. To prevent misidentifying a direct-mapped hit to an out-of-position line, it is sufficient to invalidate the cache tags corresponding to the locations that contain out-of-position lines.

Cache Miss. There are two cases to consider when the requested line is not located anywhere in the cache. The first case happens when the line to be replaced in the direct-mapped location is disposable as indicated by the d bit. In this case, the line is simply evicted from the cache. The second case occurs when the line in the direct-mapped location is not disposable. In this case, a hole has to be identified to hold this line. The primary candidate is the least-recently used (LRU) line in the OUT directory because the newly displaced line has to be entered into the OUT

directory if it is not already there. When the OUT directory has empty slots, a nearby disposable line is selected for eviction. Such a selection can be implemented by searching a word of nearby d bits with a leading 1 detector. When this fails to find a disposable line, the LRU valid entry in the OUT directory can be used as the backup candidate.

In order to avoid extraneous searching of the SHT and the OUT directory when a cache miss occurs, the d bits have to be accurately maintained. The d bit corresponding to a line is set when the line is dropped from the SHT or the OUT directory. It is reset when the line enters either directory.

2.3.2 Performance Impact

The group-associative cache has an unique combination of features that enables it to more effectively use the available cache reducing miss ratios. First of all, the lines recorded in both the SHT and the OUT directory have been referenced recently and therefore should not be evicted when a cache miss occurs. In other words, the SHT and the OUT directory help to more accurately maintain the global LRU information, thus improving the overall hit ratio.

Secondly, when a miss occurs, the line to be replaced in the direct-mapped location may be moved to another location in the cache instead of being evicted. This is similar to the victim cache approach where the replaced line or victim is always moved to the victim cache. However, unlike the victim cache which requires

a separate physical cache to hold the victims, the group-associative cache is able to effectively use the large number of holes present in the direct-mapped cache to hold the replaced lines or victims. In a group-associative cache, only a separate directory is needed to record the tags and the locations of the out-of-position lines. Therefore, a much bigger "embedded victim cache" can be built at the less cost than the original victim cache. Because the group-associative cache doesn't have the space to store data like the victim cache the group-associative cache use its underutilized frames which are identified during program execution. Moreover, the SHT and the OUT directory have a filtering effect, allowing only the more-recently used lines to enter the embedded victim cache. This selective bypassing technique helps to reduce pollution in the embedded victim cache. As far as cache pollution is concerned, the group-associative cache is superior to the fully associative cache in that a line with poor locality will be evicted relatively quickly.

Thirdly, by recording the tag and location of the out-of-position lines in the OUT directory, the group-associative cache allows these lines to be placed anywhere within the cache, and not just in a fixed alternate location as is the case in the column-associative approach. This allows the out-of-position lines to share a common pool of potential holes, thus enabling a more efficient utilization of the cache than a static partitioning scheme. In addition, by dynamically allocating the holes in response to the reference pattern, the group-associative cache is able to minimize any adverse impact on the hit ratio of the direct-mapped locations.

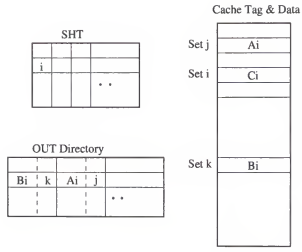


Figure 2.8. Accessing a Group-Associative Cache.

Finally, the ability to dynamically adjust the number of groups and the group associativity enables the group-associative cache to approach a fully associative cache in terms of miss ratio. For instance, in contrast to the column-associative cache which limits each set to only two lines, the group-associative cache can have groups containing anywhere from 0 to *s+1* lines, where s is the size of the OUT directory. This adaptive group size enables the group-associative cache to better capture program locality. For instance, suppose that three consecutive memory references A_i, B_i, C_i are mapped to the same cache set i. After the three references, only C_i will remain in a direct-mapped cache. In a column-associative cache, C_i will be kept in the primary location and B_i, in the alternate location. In comparison, all three most-recently used lines will remain in the group-associative cache, as illustrated in Figure 2.8. In the figure, sets j and k have been identified as potential holes and are used to hold A_i and B_i respectively.

2.4 Performance Evaluation

In this section, the methodology used to evaluate the performance impact of the group-associative cache is discussed. The evaluation is based on trace-driven simulations of workloads from both the commercial and engineering environments. Two basic metrics, miss ratio and average memory access time, are considered. Conventional and other recently proposed cache organizations are evaluated and compared against the group-associative cache.

2.4.1 Simulation Model

Separate and identical instruction and data L_1 caches are simulated. The size of the L_1 cache ranges from 8 to 64 KB with set-associativities of 1, 2, and 4. Fully-associative caches are also considered. Those on-chip L_1 caches are backed up by a 512 KB 4-way set-associative unified level 2 (L_2) cache. The line sizes of the L_1 and L_2 caches are 32 bytes and the LRU replacement policy is used in all the cases. Inclusion property is enforced between the L_2 and L_1 data caches. It is assumed that an aggressive memory system where both the memory and the L_2 cache return the critical word first and the remaining words in time for any subsequent access.

For the group-associative cache, the number of entries in the SHT and the OUT directory are varied from one-eighth to one-half the number of L_1 cache lines, (i.e., 64 to 256 lines for 16 KB cache) and from one-sixteenth to three-eighth the number of L_1 cache lines(i.e., 32 to 192 lines for 16 KB cache) respectively. In

addition, the number of sets for both the SHT and the OUT directory is varied from 1 to 8. Note that when the requested line is found in the OUT directory, it has to be swapped with the line located in the direct-mapped location. To simplify this swap, the number of sets in the OUT directory must not exceed that in the SHT. In simulations, it is assumed that an equal number of sets in both directories and an identical design for both the instruction and data L_1 caches so as to confine the total design space. The replacement policies for both SHT and OUT directory, true LRU and partitioned LRU schemes are also compared to know how replacement policy affects to the group-associative cache.

In addition, two other recently proposed schemes for improving the miss ratio of the direct-mapped cache are evaluated—the victim cache and the column-associative cache. The fully associative victim caches of one-sixteenth cache size up to 128 lines is simulated. For the column-associative cache, the secondary location is determined by flipping the highest-order index bit. As described in [3], the *rehash* bit is included with each entry of the tag array to guide the search and replacement.

In order to compute the average memory access time, the cache miss penalties at various cache levels are needed. It is estimated that these penalties using some simplifying assumptions and the general trend of current microprocessors. It is assumed that a hit in a conventional direct-mapped L_1 cache requires a single cycle. If the memory request hits in the L_2 cache, it takes 8 cycles to satisfy the request. When the request misses both the L_1 and L_2 caches, the total access delay

is 50 cycles. For set-associative caches, it is further assumed that the cycle time is lengthened by 10-20 % [30] and adjust the miss penalties accordingly.

For the victim, column-associative and group-associative caches, an extra delay is encountered when the requested data is present in an alternative location. Because of the need to swap lines in these designs and the fact that the processor pipeline is increasingly complex and difficult to turn around, it is assumed that the extra delay is 2 cycles because the bank conflict cause delays the alternative location access and swapping operation. Note that the search of the alternative locations does not add additional delay to the L_1 cache miss penalty because the L_1 cache miss can be triggered once the requested line is not present in the primary location. In other words, the delay of searching the alternative locations can be overlapped with the L_1 cache miss penalty.

2.4.2 Workloads and Traces

In order to evaluate the proposed technique with a wide variety of realistic applications, both the commercial and engineering workloads were used. For the engineering environments, 18 applications from the SPEC95 benchmark suite [55] were used.

For the commercial environment, with the help from Windsor Hsu at IBM Almaden Research Center a workload similar to the Transaction Processing Performance Council's Benchmark C (TPC-C-like) [58] was simulated. The TPC-C-like

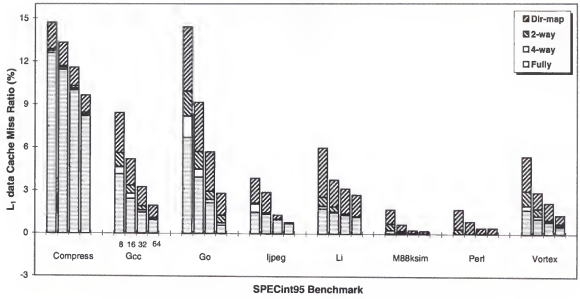


Figure 2.9. L_1 data cache miss ratio for SPECint95.

benchmark is an industry standard benchmark for measuring the performance of on-line transaction processing systems.

Sun's *Shade* tool [56] in a SPARC/SOLARIS environment to trace these SPEC95 applications was used. The standard SPEC95 input files were used. In order to avoid the initialization phase and capture the essential characteristics of these applications, the first 2 billion instructions were skipped. And the statistics by simulation were collected from another 2 billion instructions after the caches are warmed-up by 25 million instructions.

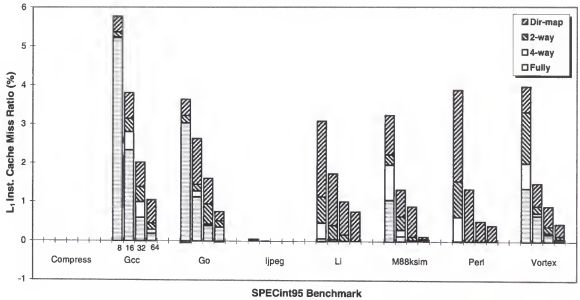


Figure 2.10. L_1 instruction cache miss ratio for SPECint95.

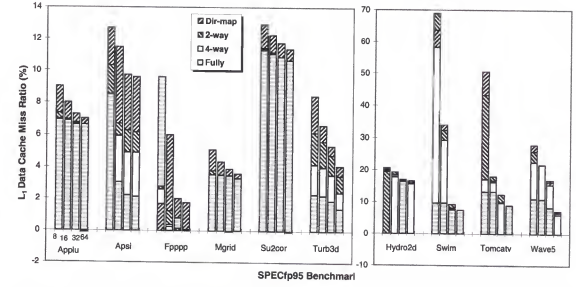


Figure 2.11. L_1 data cache miss ratio for SPECfp95 (The miss ratios of the right figure are much higher that the miss ratios of left figure).

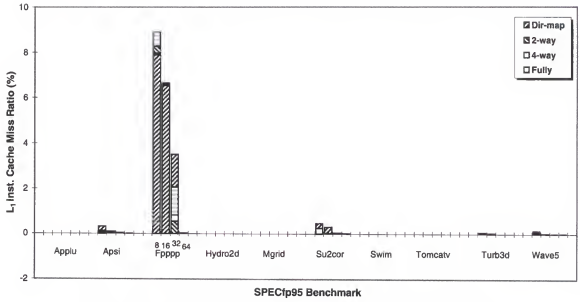


Figure 2.12. L_1 instruction cache miss ratio for SPECfp95.

2.4.3 Conventional L_1 Miss Ratios

Figure 2.9 thru Figure 2.12 summarize the data and instruction L_1 cache miss ratios for both integer applications and floating-point applications from the SPEC95 benchmark suite. Several interesting observations can be made from the figures.

First, from the SPECint95(integer programs in SPEC95 workloads) benchmark programs, some of benchmark programs, Compress, Gcc, Go, Ijpeg, Li, and Vortex, have improvement of data cache miss ratio linearly with fully associative cache all the way to 64 KB. It is suggesting that these applications suffer considerable capacity misses. And most of the SPECint95 benchmark programs have conflict misses of data cache except, Compress and Perl. Compress results show no conflict misses when set-associativity is larger than 2-way set-associativity. Perl results show that

no conflict misses when the cache size is larger than 16 KB. In instruction cache, the SPECint95 programs have capacity misses and conflict misses except, Compress, Ijpeg, and Perl. The results of Li show that it does not have capacity misses but conflict misses. The results of data cache for the SPECfp95 benchmark programs show that some workloads have very significant conflict misses. Those workloads are Apsi, Turb3d, Swim, Tomcatv, and Wave5. For example, for the workload Swim, the miss ratio of fully associative cache is 9.6 % but the miss ratio of 2-way set-associative cache is about 69 %. However, the SPECfp95(floating point programs in SPEC95 workloads) benchmark programs have very little instruction miss ratios. The simulation result of the Fpppp shows that the cache with a lower set associativity performs better than the cache with a higher set associativity when the cache size is 8 KB and 16 KB. This is due to the fact that Fpppp's working set cannot fit within smaller caches and Fpppp's instruction references are unevenly distributed across the cache sets. In the direct-mapped cache and 2-way set associative cache design have an edge over the fully associative cache by limiting the effects of cache pollution to a few sets. Most of misses are eliminated at 64 KB.

Second, the data cache of the fully associative configuration performs better than other cache configuration in all SPECint95 programs. And this kind of performance means that the data references are somewhat evenly distributed across the entire cache sets. These behaviors are also applied to data references of the most of the programs in SPECfp95. However, some of the SPECfp95 programs, which are Hydro2d, Swim, Tomcatv, and Wave5, have different reference patterns

compared to those of SPECint95 integer programs, and some of SPEC95 floating point programs. For Hydro2d, 2-way set associative cache design performs better than 4-way set associative cache and fully associative cache when the size of cache is 8 KB and 4-way set associative cache performs better than fully associative cache when the size of cache is 16 KB data cache. For Swim, direct-mapped cache design performs better than 2-way set associative cache when the size of cache is 8 KB and 16 KB. For Tomcatv, 4-way set associative cache design performs better than fully associative cache design when the cache size is 32 KB. And for Wave5, 4-way set associative cache design performs better than the fully associative cache design when the cache size is 64 KB. These are due to the fact that the data references of Hydro2d, Swim, Tomcatv and Wave5 behave in a round robin fashion which access small number of sets more frequently than other sets, it makes the smaller set associativity cache design performs better than larger set associative cache design. Also a small number of congruence group accesses make fully associative cache clear out all the way to LRU lines.

2.4.4 The Configuration of SHT and OUT directory

The performance of group-associative cache depends on how the majority of the more-recently used lines are dynamically captured using SHT and OUT directories of group-associative cache topology. Therefore, the size of SHT and OUT directory should be one of the important factors of performance of group-associative cache.

If the SHT and the OUT directory occupy a large amount of space, then the group-associative cache might be impractical. However, if the size of the directories is not enough to capture dynamic reference patterns of locality, then the miss ratio of group-associative cache will not be improved from direct-mapped cache. The goal of group-associative cache is keeping the more-recently used lines in its holes, so it can utilize the cache frames as much as possible.

The instruction and data L_1 group-associative cache miss ratios for Gcc of SPECint95 workload with 8 KB and 32 KB group associative caches are shown in Figure 2.13 and Figure 2.14.

In general, the group-associative caches with reasonably large SHT and OUT directory are able to achieve miss ratios that are consistently better than those of the conventional 4-way set-associative caches. For the instruction cache, the miss ratios are better than those of the conventional fully associative cache when the instruction cache size is 8 KB. The results suggest that the group-associative cache can indeed retain a majority of the more-recently used lines. Originally, it is expected that the group-associative cache would approaching the fully associative cache. And this expectation held true for the 8 KB and 32 KB data cache. However, for the instruction cache, the group-associative cache shows the filtering effect to the instruction reference streams. The *filtering effect* means that the cache lines in group-associative cache can be replaced faster than those of the fully associative cache. For the instruction stream, the cache lines may not be needed for a long time

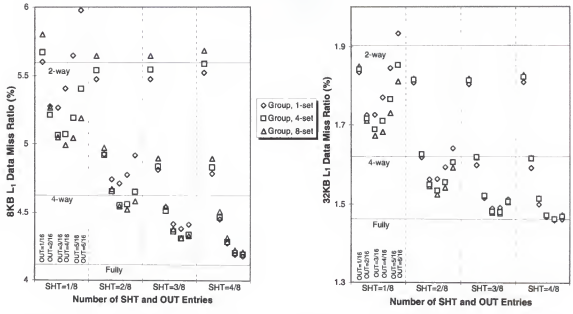


Figure 2.13. Data miss ratio with various SHT/OUT topologies(Gcc in SPECint95).

period. However, the fully associative cache keeps the lines through all the cache frames.

Notice, from the figures, that the size and topology of the SHT and OUT directory play an important role in determining the performance of the group-associative cache. First, increasing the number of SHT entries beyond three-eighth the number of cache lines hardly improves the miss ratio. However, small SHTs with entries to track only one-eighth the number of cache lines do not perform well. In this case, increasing the size of the OUT directory may even hurt the miss ratio. This is because the SHT is responsible for identifying the more-recently used lines that should be moved into the OUT directory. When the SHT is small compared to the OUT directory, it is identifying too small number of more recently used lines. As a result, some lines that have not been referenced for a while will remain in the

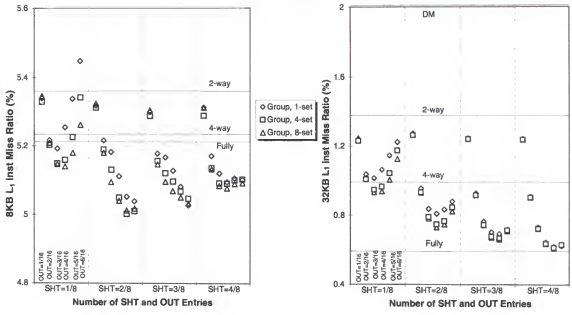


Figure 2.14. Instruction miss ratio with various SHT/OUT topologies(Gcc in SPECint95).

OUT directory. A balance between the ability to store more-recently used lines and the ability to identify them is desirable.

Second, increasing the number of entries in the OUT directory usually improves the miss ratio. However, the improvement starts to diminish when more than one-quarter of the locations are allocated to the out-of-position lines. This is due to the fact that although increasing the number of OUT directory entries does improve the hit ratio to the out-of-position lines, it also hurts the hit ratio to the direct-mapped locations.

Third, increasing the number of sets in the SHT and OUT directory from 1 to 8 has limited effect on performance for the workload Gcc. Intuitively, it is expected that a fully associative design (i.e., set=1) outperforms the set-associative design slightly. The difference should be small because the SHT and OUT directory only

helps to identify the lines that should be kept in the cache. They do not directly determine the lines that should be evicted. Therefore, even though the SHT and OUT directory become a little less accurate when they are organized into more sets, any adverse impact on the overall miss ratio should be limited.

However, notice that when the SHT is small and the OUT directory is large, and the SHT and OUT directory are organized into more sets the miss ratio actually improves. This behavior is especially pronounced with the smaller caches. A deeper analysis reveals that this unexpected behavior is a consequence of the algorithm for filling the OUT directory. Recall that the algorithm always tries to keep the OUT directory full. In other words, it always tries to hoard cache locations for storing the out-of-position lines. Depending on the reference pattern, such an aggressive policy may adversely affect the hit ratio to the direct-mapped locations. When the number of sets is increased, the hoarding phenomenon is effectively reduced because the OUT directory has more sets each of which has to be separately filled. As a result, the hit ratio to the direct-mapped positions increases without adversely affecting the hit ratio to the out-of-position lines. In short, the set-associative OUT directory is better able to adapt and adjust to the the number of locations actually needed for the out-of-position lines.

2.4.5 Improvement of Holes

The experiments about underutilized cache frames or holes, defined that more than 30 % of total cache frames are holes in direct-mapped cache in various cache sizes. If the group-associative cache can capture the dynamic reference of locality, the group-associative cache may reduce the percentage of the holes. This reduction of the holes indicates that the group-associative cache can indeed retain the majority of the more-recently used cache lines.

Figure 2.15 shows the improvement of the holes in different sizes of caches. To do this experimental simulation for the holes in group-associative cache, the SHT is three-eighth the number of L_1 cache lines (i.e., 192 lines for 16 KB cache), and for the OUT directory one-fourth the number of the cache lines (i.e., 128 lines for 16 KB cache). Note that the SHT only needs to keep track of the set-ids. This configuration is considered after investigating the miss ratios and the size of directories.

The group-associative cache can reduce the holes from 33.1 % to 15.3 % for the direct-mapped 8 KB instruction cache and for the data cache the holes are reduced from 31.6 % to 16.2 %. This hole reduction is better than 4-way set-associative instruction cache. Note that the percentage of the holes in 4-way set-associative instruction cache is 16.1 % and 15.2 % for the data cache. When the cache size is 16 KB, the holes are reduced from 33.5 % to 13.5 % for the instruction cache, and for the data cache the holes are reduced from 31.8 % to 15.5 %. The 32 and

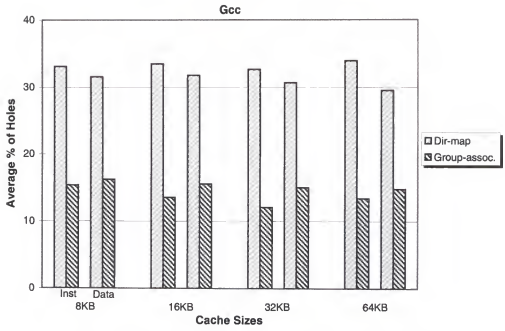


Figure 2.15. Reducing the holes (Gcc of SPECint95)

64 KB group-associative cache also reduced the holes remarkably. For the 32 KB instruction cache, the holes are reduced from 32.7 % to 12.1 % and the data cache reduces the holes from 30.7 % to 14.9 %. The numbers for 64 KB instruction cache dropped from 33.9 % to 13.4 % and the data cache reduces the holes from 29.5 % to 14.7 %. Figure 2.16 and 2.17 show the hole reduction for the SPECint95 workloads and SPECfp95 workloads. The cache size for these simulations was 16 KB data cache. From those figures, the group-associative cache can reduce the percentage of holes effectively. These results show that the group-associative cache can retain the more recently used cache lines in its cache. In general, the holes in group-associative cache is less than that of 2-way set-associative cache and about the same or less than that of 4-way set-associative cache.

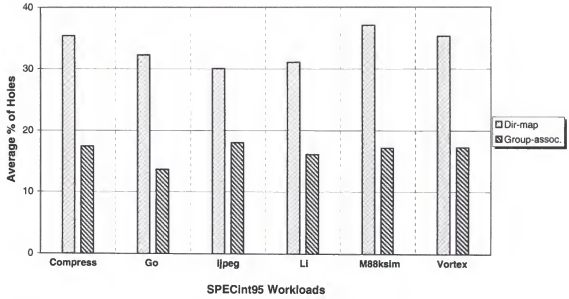


Figure 2.16. Reducing the holes for SPECint95 benchmarks (16KB data cache).

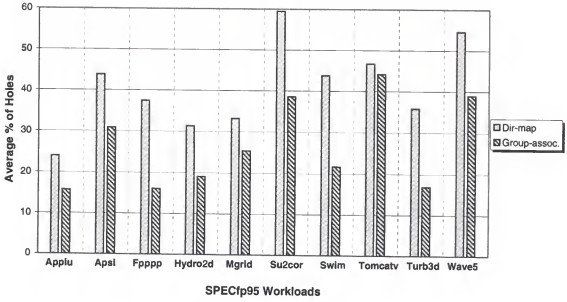


Figure 2.17. Reducing the holes for SPECfp95 benchmarks (16KB data cache).

2.4.6 Comparison with Other Cache Organizations

In this section, it has compared the performance of the group-associative cache to that of other cache organizations. Based on the results shown in Figure 2.13 and Figure 2.14, two SHT sizes — $\frac{2}{8}$ and $\frac{3}{8}$ considered. When the number of entries in the SHT is two-eighth the number of cache lines, the OUT size is $\frac{3}{16}$ and $\frac{4}{16}$. When the SHT size is $\frac{3}{8}$, the number of entries in the OUT directory is four-sixteenth and five-sixteenth the number of cache lines. After investigating configurations of the SHT and OUT directory, SHT and OUT directory are designed with 8 sets. For group-associative cache, (a, b) represents that a is the ratio of SHT directory to the cache sets and b represents the ratio of OUT directory to the cache size.

Observe in Figure 2.18 that the selected group-associative caches almost always achieves a lower miss ratio than the victim and column-associative caches. For instance, for the 16 KB data caches, the miss ratios for the 32-line victim cache and the column-associative cache are about 26 % and 36 % higher than that of the $(\frac{3}{8},\frac{4}{16})$ group associative cache for the workload Gcc in SPECint95. For the 16 KB instruction cache, the numbers are 33 % and 29 % respectively.

Among the enhanced caches, group-associative cache $(\frac{3}{8},\frac{5}{16})$ has the lowest miss ratio followed by group-associative cache $(\frac{3}{8},\frac{4}{16})$. The column associative cache, on the other hand, only achieved a miss ratio that was close to that of the 2-way set-associative cache. Figure 2.19 and 2.20 show the miss ratios of 16 KB data cache for the rest of SPEC95 workloads. The figures show the group-associative cache

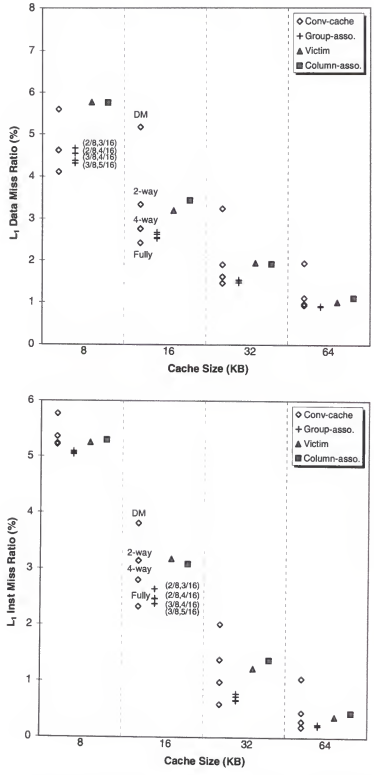


Figure 2.18. Gcc of SPECint95 workloads miss ratio with conventional and enhanced caches.

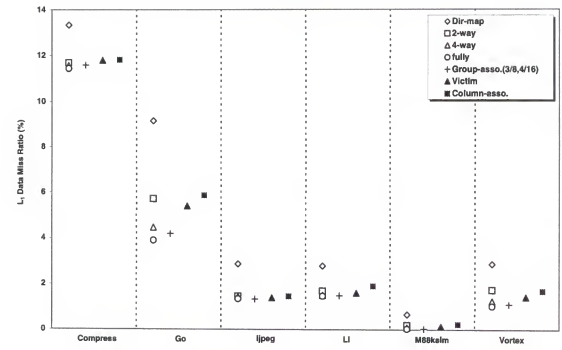


Figure 2.19. SPECint95 workloads miss ratio with conventional and enhanced caches (16KB data cache).

can achieve a lower miss ratio than the miss ratios of victim cache and column-associative cache.

There are two fundamental reasons as to why the group-associative cache is able to achieve a better overall miss ratio. First, the group-associative cache has the ability to capture extra hits to the out-of-position lines. Second, it is able to do this with minimal adverse impact on the hit ratio of the direct-mapped locations. This is illustrated in Figure 2.21 in which the hit ratio to the direct-mapped and the alternative locations for the victim, column- and group-associative caches are plotted. As expected, the hit ratio to the direct-mapped locations remain unchanged for the victim cache. This hit ratio is reduced for both the column- and group-associative caches. However, due to the flexible locations for the out-of-position

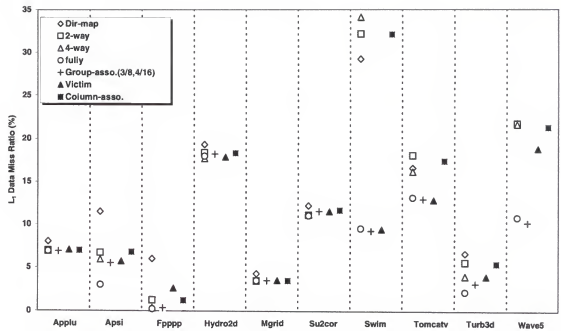


Figure 2.20. SPECfp95 workloads miss ratio with conventional and enhanced caches (16KB data cache).

lines and the adaptive sharing of these locations, the reduction in hit ratios to the direct-mapped locations is very minimal for group-associative caches.

Furthermore, notice that the hit ratio of the alternative locations is noticeably higher for the group-associative cache. For the 32 KB caches and same workload, the hit ratios of the out-of-position lines are 1.28 % and 1.42 % for the $(\frac{2}{8},\frac{3}{16})$ and $(\frac{3}{8},\frac{4}{16})$ group-associative caches. The hit ratio of the alternative locations in the column-associative cache is 1.06 % while the hit ratio of the victim cache with 64 lines is 0.72 %. In comparison with the victim cache, the higher hit ratio of the group-associative cache comes from the efficient utilization of holes to build the bigger embedded victim cache plus the ability to selectively bypass this embedded victim cache. When compared to the column-associative cache, the group-associative cache

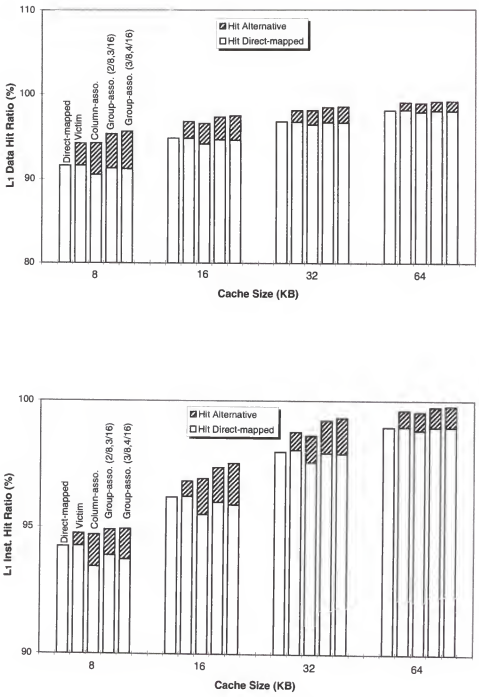


Figure 2.21. Impact on direct-mapped cache hit ratio (Gcc of SPECint95).

prevails because of its ability to dynamically adjust the number of groups and the group associativity.

In addition to the overall hit/miss ratios, the different delays in accessing the direct-mapped and the alternative locations should also be considered in evaluating the performance of the various cache organizations. The average memory access time is used as the performance metric. Recall that in the simulation model, it is assumed that a hit to an alternative location in the victim column- and group-associative caches takes 3 cycles. This is because given extra buffer for the cache, the swap doesn't need to involve the processor which does other work while the cache becomes available again and, if this is the case half of the time, then the time wasted by a swap is one cycle [3]. It is assumed that the set-associative design lengthens the cycle time by 10-20 %.

Figure 2.22 summarizes the average memory access time for the various cache organizations in terms of the number of cycles for the workload, Gcc. Note that the set-associative results are normalized to the direct-mapped cycle time. Due to the longer cycle time with the set-associative caches, the conventional caches in general do not perform as well as the other cache organizations. The group-associative caches generally have the shortest average memory access time. For instance, for the 16 KB data cache, the best average memory access time for the group-associative, victim, and column-associative caches are 1.29, 1.33, and 1.35 respectively for the workload Gcc. The numbers are 1.22, 1.26, and 1.27 for the 16 KB instruction cache. Figure 2.23 and 2.24 show the average memory access

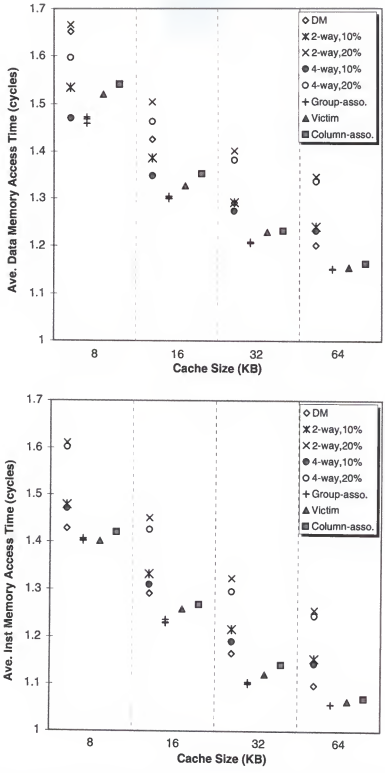


Figure 2.22. Average memory access time comparison (Gcc in SPECint95).

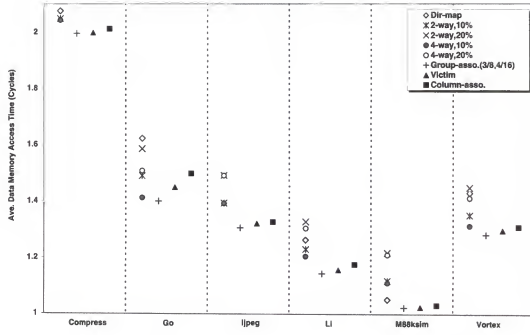


Figure 2.23. Average memory access time comparison for SPECint95 benchmarks (16KB data cache)

time for the rest of SPEC95 workloads. The cache size is 16 KB data cache. From the figures, in general, the group-associative cache can achieve the fastest memory access time.

Table 2.4. Extra space calculation (16KB L_1 cache)

	TOTAL BIT	SHT	OUT	d-BIT	TAG	DATA
Group.	6.5K	6×192	38×128	1×512	0	0
Victim.	9.3K	0	0	0	35×32	256×32

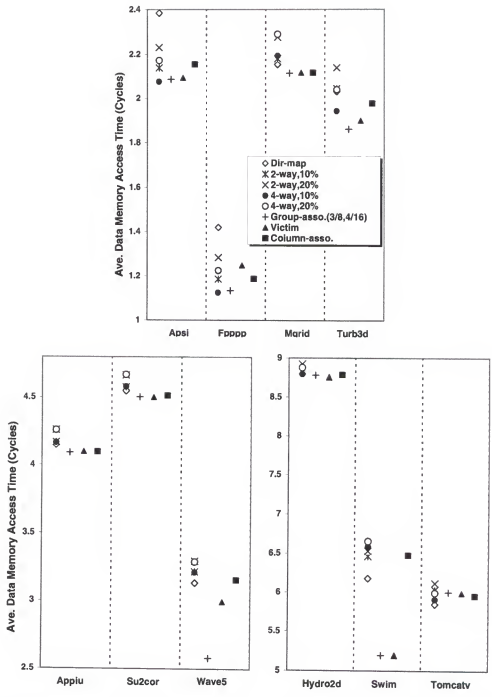


Figure 2.24. Average memory access time comparison for SPECfp95 benchmarks (16KB data cache)

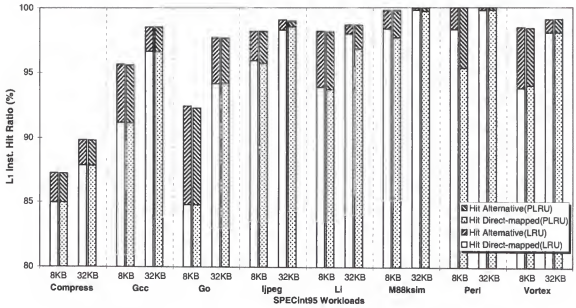


Figure 2.25. The comparison between true LRU and PLRU (SPECint95).

The group-associative cache requires additional chip area to implement the SHT, the OUT directory, and the *disposable* bit. Consider the 16 KB ($\frac{3}{8}$,$\frac{4}{16}$) group-associative cache with 8 sets and a 40-bit physical address space, the extra space needed is about 6.5 Kilo-bits (Kb). Without accounting for the peripheral logic, the additional area required is about 4.3 % of that taken up by the cache. For the victim cache, it requires about 6.25 % of that taken up by the cache. Table 2.4 shows the calculation.

2.4.7 Partitioned LRU replacement of SHT and OUT-directory

It has been compared between true LRU replacement policy with a simpler partitioned LRU (PLRU) replacement scheme [54]. The PLRU scheme is a simpler

mechanism to implement LRU replacement algorithm. The Partitioned LRU(PLRU) scheme uses bits to distinguish MRU and LRU side from the entry lists. Thus, PLRU scheme need *n-1* bits to build PLRU replacement algorithm for *n* entries.

Figure 2.25 and Figure 2.26 show the results of the comparison between true LRU and PLRU scheme for the group-associative cache. From the result, it is shown that the difference between two replacement scheme does not affect the miss ratio of the group-associative cache much. This is because the SHT and OUT directory only help to distinguish the lines that should be kept in the cache. They do not directly determine the lines that should be evicted. Therefore, even though the SHT and OUT directory become a little less accurate when they are organized into PLRU replacement scheme, the adverse impact on the overall miss ratio is limited. Some of the workloads, Li, M88ksim, and Perl, show that the PLRU scheme does the hit ratio to the direct-mapped cache lower and increase the hits on the alternative locations. This is because that the lower accuracy of the replacement scheme makes the lines of the direct-mapped location replace quickly and stay in the alternative location. Thus, the total hit ratio to the primary cache is not affected but the hit on the direct-mapped location is decreased.

From the result, despite integer programs and floating point programs, the miss ratio difference between true LRU and PLRU scheme is less than 0.1 %.

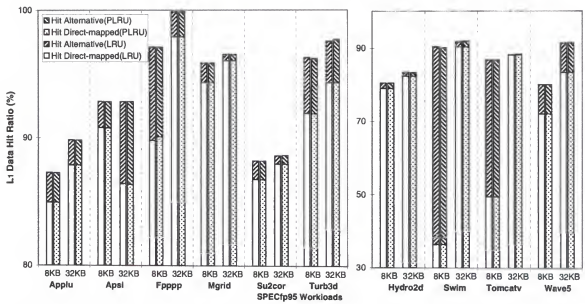


Figure 2.26. The comparison between true LRU and partitioned LRU (SPECfp95).

2.4.8 Performance with Embedded Victim Cache

Figure 2.27 shows the miss ratio comparison between the victim cache and the embedded victim cache which is built on the group-associative cache. To see the performance of the embedded victim cache, the group-associative cache which, has only OUT-directory to hold the more recently used cache lines, is simulated. The replaced cache lines from its direct-mapped location is placed to the LRU entry of the OUT-directory. Thus, the OUT-directory can hold the recently replaced victim cache lines in the primary cache. When the OUT-directory has empty slots, a nearby location is selected for eviction like the group-associative cache.

From the figure, the embedded victim cache with four-sixteenth the number of L_1 cache lines OUT-directory has the lower miss ratio than that of the victim cache.

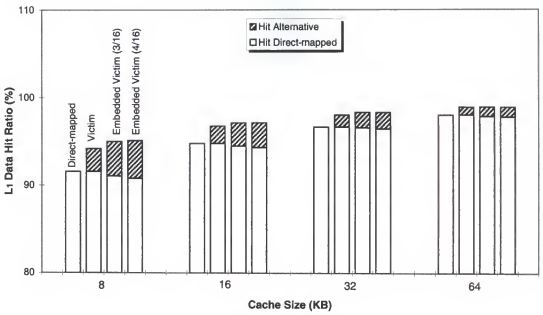


Figure 2.27. Miss ratio comparison between victim cache and embedded victim cache(Gcc of SPECint95).

When the embedded victim cache size is bigger the hit ratio on the direct-mapped location is reduced while the hit on the alternative location is increased. When the cache size is 16 KB, the hit ratio for the embedded victim cache is 97.2 % with one-fourth the number of L_1 cache lines OUT-directory. The victim cache with one-sixteenth the number of L_1 cache lines victim cache has 96.8 %. Note that the embedded victim cache needs less space than the victim cache since the embedded victim cache stores only the tags and set-ids on its OUT-directory.

2.4.9 The result of TPC-C-like benchmark

TPC-C-like benchmark is an industry standard benchmark to compare the performance of systems. TPC-C-like simulates a complete computing environment

where a population of terminal operators execute transactions against a database. The benchmark is centered around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. While the benchmark portrays the activity of a wholesale supplier, TPC-C-like is not limited to the activity of any particular business segment, but, rather represents any industry that must manage, sell, or distribute a product or service.

Figure 2.28 plots the average percentage of holes that exist in various cache configurations after each memory reference during the execution of TPC-C-like. The results show that a very significant number of holes exist in both the instruction and data caches. For the direct-mapped data caches, between 37.5% and 42.6% of the cache are holes. The corresponding ranges for the 2-way and 4-way set associative caches are about 27–33% and 20–24% respectively. Compared with the data caches, the instruction caches have fewer holes. About 34–37% of the direct-mapped instruction caches are holes. The corresponding numbers for the 2-way and 4-way set associative caches are about 24–27% and 17–18% respectively.

Observe in Figure 2.29 that for TPC-C-like, the group-associative cache is able to achieve a miss ratio that is comparable to or better than that of the 4-way set-associative cache. In certain cases, the miss ratio approaches that of the fully associative cache. The results suggest that the group-associative cache can indeed retain a majority of the more-recently used lines. Further confirming this effect, it

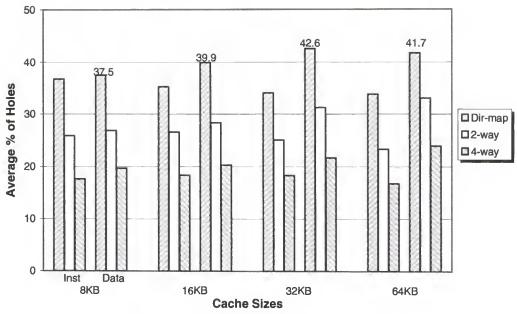


Figure 2.28. Average percentage of holes (TPC-C-like).

is found that the average percentage of holes in the 8 KB, 16 KB, 32 KB, and 64 KB data caches has improved respectively from 37.5 %, 39.9 %, 42.6 %, and 41.7 % in the direct-mapped design to 18.2 %, 18.7 % 19.9 %, and 19.0 % in the $(\frac{3}{8},\frac{4}{16})$ group-associative caches. With a slightly bigger OUT directory $(\frac{5}{16})$, the percentage of holes is further reduced to 15.9 %, 15.5 %, 15.6 %, and 14.5 % respectively.

In addition, the selected group-associative caches achieve lower miss ratio than the victim and column-associative caches and the margin can be very sizable. For instance, for the 32 KB caches, the miss ratios for the victim cache and the column-associative cache are about 10.7 % and 11.1 % higher than that for the $(\frac{3}{8},\frac{4}{16})$ group associative cache. Also, for the data cache, the miss ratio for the victim cache and the column-associative cache is about 28 % and 26 % higher than that for the $(\frac{3}{8},\frac{4}{16})$ group associative cache. Among the group-associative caches, those with

bigger SHT and OUT directories perform better. For example, $(\frac{3}{8}, \frac{5}{16})$ has the lowest miss ratio followed by $(\frac{3}{8}, \frac{4}{16})$. The victim and the column associative caches show very similar miss ratio that is close to that of the 2-way set-associative cache.

Figure 2.30 summarizes the average memory access time for the instruction and data references with various cache organizations. Note that all the results are normalized to the direct-mapped cycle time. Due to the longer cycle time with the set-associative cache, the conventional cache does not perform as well as the other cache organizations. Only the 4-way set associative design with an optimistic 10% cycle time degradation shows performance comparable to the victim and the column-associative caches. The group-associative cache has the shortest average memory access time. For instance, for the 32 KB cache, the best average instruction memory access time for the group-associative, victim, and column-associative caches is 1.52, 1.55, and 1.57 respectively. Also, the best data memory access time for the group-associative, victim, and column-associative caches is 2.44, 2.54, and 2.55 respectively.

2.5 Previous Works

A general strategy to simultaneously attain a fast cache access time and a high hit ratio is to have two cache access paths. A fast path is used to achieve fast access time for the majority of memory references while a relatively slow path is

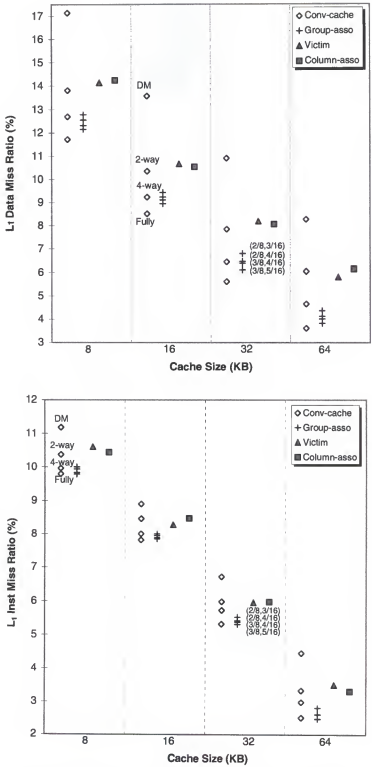


Figure 2.29. Miss ratio comparison (TPC-C-like).

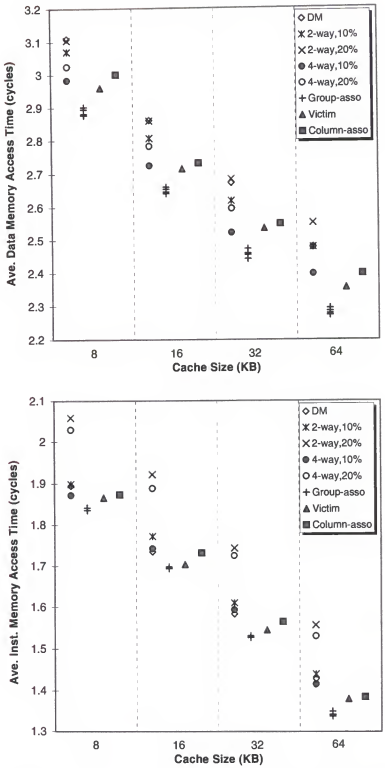


Figure 2.30. Average memory access time comparison (TPC-C-like).

used to boost the effective hit ratio. Two broad categories of such techniques can be distinguished.

The general idea in the first category is to decouple the tag and data paths in cache access so that, for the majority of memory references, the fast data array access and line selection can be carried out independently of the slow tag array access and comparison. Examples of techniques in this category include the MRU cache [10], the line-ID prediction scheme [7, 42], the partial-tag matching technique [41], the Direct-mapped Access Set-associative Check (DASC) cache [50], the difference-bit directory [31], and the alternative tag path method [46].

Techniques in the second category access a direct-mapped cache sequentially more than once in order to achieve a fast access time in the first access and a high hit ratio as a whole. Examples of this include the hash-rehash cache [2] and the column-associative cache [3]. A way to extend the column-associative cache to include multiple alternative locations is described in [13, 62].

A number of methods have been proposed to reduce cache conflict misses. One technique is to build a small buffer or victim cache to hold the lines that have been recently evicted from the cache [29]. The HP-PA7200 uses a small on-chip FIFO buffer called the *assist* cache, in addition to a direct-mapped L_1 cache, to ensure that the very recently used data will not be susceptible to conflict misses [36]. In [27, 47], a small fully associative buffer is proposed for holding the lines that exhibit poor temporal locality so as to prevent them from entering and polluting the primary

direct-mapped cache. Another approach to reducing conflict misses is to use better hashing or mapping functions [18, 49, 53].

CHAPTER 3
DATA PREFETCHING

3.1 Introduction

Prefetching refers to the fetching of data from the lower level of memory hierarchy before a reference to the data actually happens, so that the processor doesn't generate cache misses to the prefetched blocks. Prefetching attempts to hide miss penalties by overlapping memory access with instruction execution. The prefetching schemes can broadly be classified into two groups, software-directed approaches [12, 19, 34, 43, 44, 48] and hardware-based approaches [7, 12, 29]. Software-controlled prefetching schemes rely on compiler technologies or possibly user directed static program analysis to selectively insert prefetch instructions based on its information of program behaviors. The prefetch instruction initiates a fetch of the designated block(s) into cache memory during the execution and its action happens before the normal access to the block(s). The success of software-directed prefetching depends primarily on identifying and inserting prefetch instructions only for those accesses that are most likely to generate cache misses. Once a potential cache miss has been identified, the software prefetching scheme insert a prefetch instruction.

On the other hand, hardware-based prefetching attempts to fetch block(s) ahead based on dynamic reference pattern during program execution. For hardware-based prefetching scheme, there are several prefetching topologies. The *prefetch always* topology fetches a sequential reference pattern in order of increasing memory address initiating a prefetch to the next line when the current block is accessed [52]. The *prefetch-on-miss* and *tagged-prefetch* topologies are also based on the idea of prefetching the next blocks based on a reference to the current line, however, those schemes do the prefetch when the cache miss and/or hit on the prefetched blocks [29, 53]. The data prefetching scheme, which prefetches on miss and hit on the prefetched data, is called *sequential* data prefetching in this dissertation research. Moreover, there are some prediction based prefetching topologies, which decide whether a block needs to be prefetched or not based on decision making information. The prefetch schemes using *stride* information [11], *Markov chain* [28] and *filtering* [45] belong to this topology. These prefetch schemes decide whether the cache block is prefetched or not, and/or which block(s) need to be prefetched by the information of stride of reference pattern or Markov access pattern.

The *Stride Directed Prefetch (SDP)* scheme uses the relationship between the vector stride distance and the cache block size to direct prefetching. SDP scheme uses a history table to calculate the stride distance of array or vector accesses made from within the body of program loops. *Markov Predictor* [28] scheme uses the prediction addresses which are stored in *prefetcher*. The Prefetcher maintains the next addresses for a each miss. And when a miss happens for a specific address,

prefetcher finds the next addresses to prefetch lines from lower level to higher level with missed one. The prefetcher information is based on miss address reference strings, i.e., miss reference string shows the next miss when a miss happens. The next misses of a certain address are collected to predict the prefetching lines. The *stream buffers* [29] are design to prefetch sequential streams of cache lines, independent of program context. Stream buffers are allocated on first level cache misses. If any stream buffer entry has a hit which misses first level cache, the line is taken from the stream buffer. The entries below the one removed are all shifted to the head of the buffer, and prefetches are launched to sequential consecutive cache line addresses to fill the vacancies that open up the bottom part of stream buffer. If there is no match when a cache miss happens, a stream buffer is allocated to the new stream. The stream buffer scheme is also employed by a *filtering* [45] mechanism which waits for two consecutive first level cache misses to the sequential cache line addresses before a stream is allocated in the stream buffer.

Even though software prefetch schemes do not use a separate place to put the prefetched line since the software scheme use prefetch hint information which is analyzed under compilation or user direction. These hardware prefetch schemes generally use a separate space to place the prefetched line. The stream buffer and filtered prefetch schemes are using a stream buffer which has FIFO entries to prevent polluting the primary cache, so no other cache line which is not actually accessed can be located in main cache. The Markov Predictors scheme also use separate space to place the prefetched cache lines, which is called *Prefetch Buffer*. The Prefetch

buffer places the prefetched line so the prefetched line(s) do not pollute the primary cache. When a prefetched line is a hit in the prefetch buffer the Markov Predictor scheme moves the prefetched line to first entry of FIFO prefetch buffer instead of moving it to the primary cache. A match in the stream buffer causes the line to be moved to the main cache.

Unlike other prefetching, the data prefetching in group-associative cache can be handled without separate space to place the prefetched cache lines. The reason for this is that the group-associative cache can identify the less recently used cache frames which can be used as separate space to place the prefetched lines. The space, of the underutilized cache frames, provides a bigger space to place prefetched lines. This can improve the performance of prefetching to group-associative cache can beyond that of the other prefetch schemes which use the extra space.

3.2 Statement of Problem

One of the major issues in data prefetching is that the prefetched data may not actually be used and will pollute the cache if they are brought in. Every prefetch operation involves removing some other, already resident cache line from cache memory and/or one or more lines of extra space dedicated to prefetched data. If the removed cache line is referenced sooner than the prefetched line, this prefetch would increase the number of cache misses and cause a loss of bandwidth to bring the line to cache memory. Thus, the place to put a prefetched block is one

of the major factors of prefetching efficiency. If prefetched blocks replace existing heavily-referenced blocks, then the prefetching makes the cache miss ratio higher and valuable system resources would be wasted. This phenomena is called *cache pollution.*

The cache pollution may happen for every data prefetching scheme since all the prefetched lines might not be used by a processor before the prefetched line is replaced. To reduce this cache pollution, the software schemes have tried to get more accurate information to prefetch and hardware schemes have used extra space to prevent this phenomena. The cache pollution has two effects on the system performance. The first one is the cache hit ratio which is lower since the heavily referenced cache line can be replaced, the extra misses to these lines cause a loss of cache hit ratio. The second one is extra bandwidth which needs extra data transfer which may be wasted.

Figure 3.1 and Figure 3.2 show cache miss ratios and extra memory references with several data prefetching algorithms. This experiment uses the workload Gcc in SPECint95. Figure 3.1 shows the miss ratio and extra reference when no prefetch buffer is used, and Figure 3.2 shows the miss ratio and extra traffic when a prefetch buffer is used. The sequential prefetching scheme, filtering stream buffer prefetching scheme and the stride data prefetching schemes are used to see the effects of cache pollution. For this experiment, an eight-entry *history table* to identify the existing sequential access blocks is used for the filtering sequential prefetching scheme and

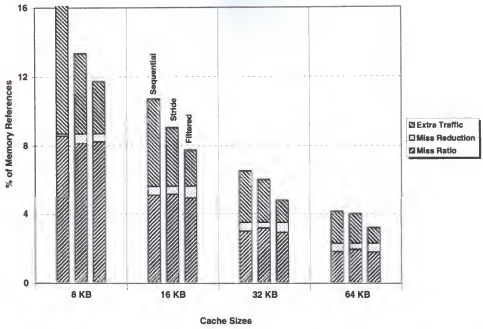


Figure 3.1. Cache miss ratios and extra traffics (Gcc of SPECint95, no Prefetch Buffer)

a 256-entry *stride prefetch table* is used to identify the strides for the memory references. The miss ratio as well as the extra reference generated by data prefetching are evaluated. The extra reference represents a fraction of the total memory references and are the references which are not used by the processor. The extra references are brought from lower memory hierarchy; but, these transfers from lower level memory hierarchy are not used for the accesses. The miss ratio element is the miss ratio with data prefetching, and the miss reduction element is the difference between the miss ratio without data prefetching and the miss ratio with data prefetching. Hence, the addition of the miss ratio and miss reduction comes up with the miss ratio without data prefetching. The prefetched data locates in its direct-mapped position without prefetch buffer. The *prefetch buffer* topology uses separate space

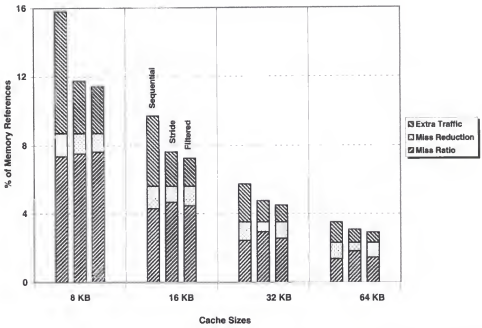


Figure 3.2. Cache miss ratios and extra traffics (Gcc of SPECint95, with Prefetch Buffer)

to place the prefetched cache line. When a hit occurs on the prefetched cache line the line will be located in top of the prefetch buffer but not in the main cache. The prefetch buffer topology, which adds a separate buffer to direct-mapped cache, has more improvement than direct-mapped cache with data prefetching. This is because the separate space reduce the pollution to the primary cache.

From the Figure 3.1, the miss ratios for the various cache size are reduced when the data prefetching schemes are applied. The filtered prefetch scheme miss ratio is reduced from 8.7 % to 8.2 % when cache size is 8 KB. However, the extra memory references are increased by about 3.0 %, which is 35 % more memory traffic than that of the no-prefetching scheme. Note that no-prefetching represents the cache scheme without prefetching. Therefore, the data prefetching increased the memory

references about 35 %, and those prefetched lines are not used and pollute the cache. When the cache size is 16 KB, the filtered prefetching scheme can reduce the miss ratio from 5.6 % to 4.9 %, with 2.1 % of extra memory traffic. The filtered prefetching scheme increases the memory traffic by 38 % when the cache size is 16 KB. When the cache size is 32 or 64 KB, the miss ratios are reduced from 3.5 % to 2.9 % and from 2.3 % to 1.8 % respectively. The extra memory references compared to no-prefetching scheme are increased about 37 % and 39 % respectively.

Therefore, data transfers are increased more than 35 %, these transfers were not used. However, these lines may pollute the cache when the filtered data prefetching scheme is applied.

The stride prefetching scheme can also reduce the cache miss ratio from 8.7 % to 8.1 % with 4.6 % extra memory references when the cache size is 8 KB. This extra memory traffic generates about 53 % more unused data references than the no-prefetching scheme. When the cache size is 16, 32, and 64 KB the miss reductions are 0.47 %, 0.32 %, and 0.34 % respectively. The extra memory traffics for 16, 32, and 64 KB are 3.45 %, 2.49 %, and 1.69 % respectively. This extra memory traffic increases the bandwidth about 60 % more than that of no-prefetching scheme.

Compared to filtered prefetching and stride prefetching, the sequential prefetching generates more traffic than other data prefetching schemes. The 8 KB data cache sequential prefetching reduces the miss ratio from 8.7 % to 8.6 % with 8.6 % extra traffic. This extra traffic increases the bandwidth to about 98 % of that of no-prefetching scheme. The 16, 32, and 64 KB, miss ratios are reduced 0.51 %, 3.01

%, and 0.47 % respectively. The memory traffic which is not used is increased 5.1 %, 3.0 %, and 1.8 % respectively.

As discussed above, when the data prefetching schemes are used, the extra data transfers, which are not used but replaced by another line, pollute the cache and this pollution affect the performance of the cache memory.

3.3 Handling Data Prefetching Using Group-Associative Cache

The ability to dynamically identify the underutilized cache frames or *holes* in group-associative caches provides another advantage that can accommodate data prefetching with minimum cache pollution. In order to prevent cache pollution, a separate stream buffer [29] or a prefetch buffer [28] is designed to hold the prefetch data in conventional caches. In contrast, the group-associative cache is able to effectively use the large number of holes presented in the direct-mapped cache to hold the prefetched blocks. This eliminates the need for a separate physical buffer and enables a bigger "embedded" prefetch buffer. This dissertation research does not introduce a new algorithm of data prefetching scheme but a new way of placement for prefetched line is considered. Several data prefetching algorithms are considered to simulate the performances. Whenever a miss occurs, the *sequential prefetch* method prefetches the next sequential lines if it is not already in the cache. Upon the first access to a prefetched line, it triggers another prefetch of the following sequential line [52]. The *filtered* sequential prefetch method, on the other hand,

starts prefetching the next sequential line on a miss only when a previous sequential access pattern has been identified [45]. In general, with the filtering technique, the useless prefetch may be reduced at the cost of lower cache hit ratio improvement.

The way that the prefetched lines are being handled in the group-associative cache is straightforward. When the direct-mapped location for a prefetched line consists of a disposable line, the prefetched line simply overwrite the disposable block and is marked as disposable. In the case that the direct-mapped location is occupied by a non-disposable line, then a hole must be identified to be evacuated for the prefetched line. The mechanism to identify a hole is the same as in the case of a regular cache miss occurring and the direct-mapped location has a non-disposable block. In this case, the prefetched block tag is inserted in the middle of the LRU sequence in the OUT directory so that the impact to the existing out-of-position lines and the longevity of the prefetched line in cache can be balanced.

With regard to data prefetching, it is interesting to find that the victim and column-associative cache can also reduce cache pollution by using additional victim cache and the alternative locations respectively to hold the prefetched lines without an extra prefetch buffer.

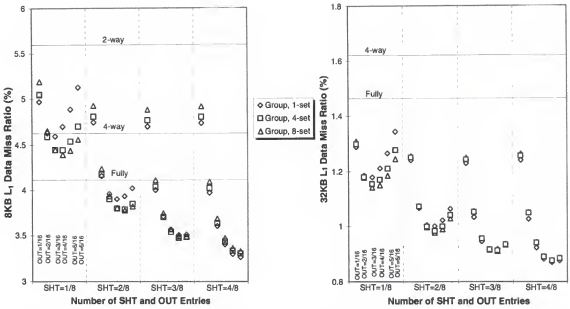


Figure 3.3. Data cache miss ratio with various SHT/OUT topologies with data prefetch(Gcc in SPECint95).

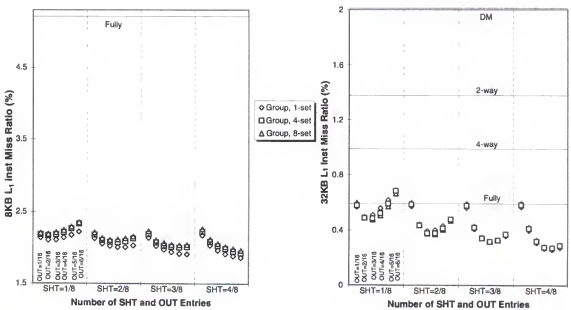


Figure 3.4. Instruction cache miss ratio with various SHT/OUT topologies with data prefetch(Gcc in SPECint95).

3.4 Performance of Data Prefetching

3.4.1 The Configuration of the Group-Associative Cache for Data Prefetch

The size and configuration of the SHT and OUT directory are the primary parameters which impact the performance of the group-associative cache. When the data prefetching is not applied in a group-associative cache, the goal of the size and configuration of the SHT and OUT directory may capture the dynamic locality behavior of the program execution. With data prefetching schemes, the size and organization of the both directories are still important factors in the performance of the group-associative cache with applying data prefetching. Thus, the simulations with various size and configuration of the SHT and OUT directory need to be completed when a data prefetching scheme is applied to the group-associative cache.

The number of disposable lines is dependent on the number of entries in the SHT and OUT directory. When there is a very small number of disposable lines in group-associative cache, there is a small chance the prefetched line is located in a disposable location. These disposable cache frames are the locations to place the prefetched lines without interrupting the cache lines which are in OUT directory. Therefore, the relationship between the number of disposable cache lines and the performance of the group-associative cache with data prefetching is an interesting investigation. This investigation of the size and configuration of the SHT and OUT

directory may produce the best performance when a data prefetching scheme is applied to group-associative cache.

When the direct-mapped position for the prefetched cache line is not disposable, the line should be placed in out-positioned location. Those lines which are prefetched and placed in out-position, need to be recorded in one of the OUT directory entries. Since the prefetched line is placed into OUT directory without actual access of the line, the prefetched line can be stored in any position of the sequence of the OUT directory. The study of the positions of prefetched line in OUT directory needs to be done to see the impact of group-associative cache by the positions.

Figure 3.3 and 3.4 show the miss ratios of various group-associative cache after applying the sequential prefetching. Compared to the group-associative cache without prefetching, the behavior of miss ratio patterns depending on SHT and OUT directory is similar to both cases. Thus, the reasonable size of SHT is two-eighth and three-eighth the number of L_1 cache lines and the OUT directory is three-sixteenth and four-sixteenth the number of L_1 cache lines.

3.4.2 The Comparison with other cache Topologies

The performance of data prefetching in group-associative cache depends on how the cache topology identifies the underutilized cache frames which can hold the prefetched lines. If the group-associative cache can reduce the pollution, the miss ratio of the group-associative cache with data prefetching should be lower than that

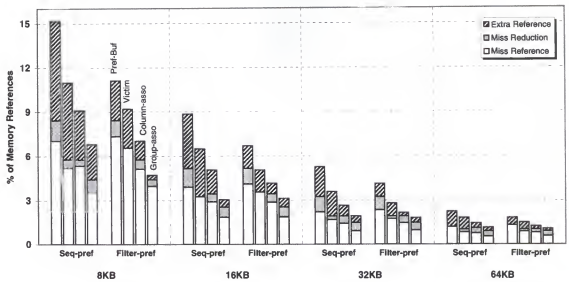


Figure 3.5. Cache miss ratios and extra traffics (Gcc of SPECint95)

of group-associative cache without data prefetching. The extra reference using data prefetching is the major performance indicator to see how much the cache pollution is decreased. In the result shown in Figure 3.5, the group-associative cache improves the miss ratio with sequential and filtering data prefetching schemes while the other cache topologies suffer from pollution of the data prefetching. The workload Gcc of SPECint95 is used for this simulation. The group-associative cache $(\frac{3}{8}, \frac{4}{16})$ is used.

Based on the Figure 3.5, the group-associative cache can reduce the miss ratio from 4.8 % to 4.0 % with 2.6 % extra memory traffic when the cache size is 8 KB and the sequential data prefetching scheme is applied. However, the direct-mapped cache with prefetch buffer reduces the miss ratio from 8.7 % to 7.3 % with 7.1 % extra memory traffic. Thus, the group-associative cache can reduce the memory traffic which pollutes the cache from 7.1 % to 2.7 %, which is about 65 % reduction of the memory traffic which generates the cache pollution. When the filtered prefetching

scheme is applied to both group-associative and direct-mapped caches with prefetch buffer, the group-associative cache reduces the miss ratios from 4.8 % to 4.0 % with 1.1 % of extra memory traffic. However, the direct-mapped cache with prefetch buffer reduces the miss ratios from 8.7 % to 7.6 % with 2.7 % extra memory traffic. The reduction of the extra memory traffic which generates the cache pollution is reduced from 2.7 % to 1.1 %, which is about 60 % reduction of prefetching which generates cache pollution.

When the cache size is 16, 32, and 64 KB, the same phenomena occurs. When the cache size is 16 KB the group-associative cache reduces the memory traffic remarkably which pollutes the cache about 68 % for the sequential data prefetching scheme and 63 % for the filtered data prefetching. When the sequential data prefetching scheme is applied to the 32 and 64 KB cache, the group-associative cache can reduce the extra memory traffic remarkably generating cache pollution of 77 % and 73 % respectively. These numbers are the percentage of with extra memory traffic of direct-mapped cache compared with the prefetch buffer. When the filtered data prefetching scheme is applied the extra memory traffic reduction is about 62 % for 32 KB cache and 56 % for 64 KB cache.

In general, the group-associative cache can reduce the extra memory traffic which generates cache pollution 50 % more than that of direct-mapped cache with prefetch buffer in various cache sizes.

3.4.3 The Results of SPEC95 Workloads

Figure 3.6 thru Figure 3.10 plot the data miss ratios for the SPEC95 benchmark suite. Observe that the SPEC95 programs, especially those that are floating-point intensive, show vastly different cache behavior. In these figures, only two configurations are considered, namely $(\frac{2}{8},\frac{3}{16})$ and $(\frac{3}{8},\frac{4}{16})$, for the group-associative cache. The effect of applying filtered sequential prefetch is presented.

The group-associative cache consistently achieves a miss ratio that is equal to or better than that of the 4-way set-associative cache for the SPEC95 programs. The miss ratio improvement is especially significant for applications such as Gcc, Go, Tomcatv, Turb3d, Vortex, and Wave5 which exhibit high conflict misses. For instance, Turb3d's miss ratio with a 32 KB cache is reduced from 5.5 % and 3.7 % with the direct-mapped and 4-way set-associative designs respectively to 2.6 % with the $(\frac{2}{8},\frac{3}{16})$ group-associative cache. On the other hand, since most of the conflict misses for Compress, Hydro2d, and Su2cor can be eliminated by the 2-way set-associative design, the benefit of the group-associative cache is not as dramatic. Observe also that the difference in miss ratio between the two group-associative caches is very minor for these SPEC95 applications.

Notice that the cache behavior of Tomcatv and Wave5 is unusual in that the direct-mapped, set-associative, and group-associative designs may slightly outperform the fully-associative cache in certain configurations. This is due to the

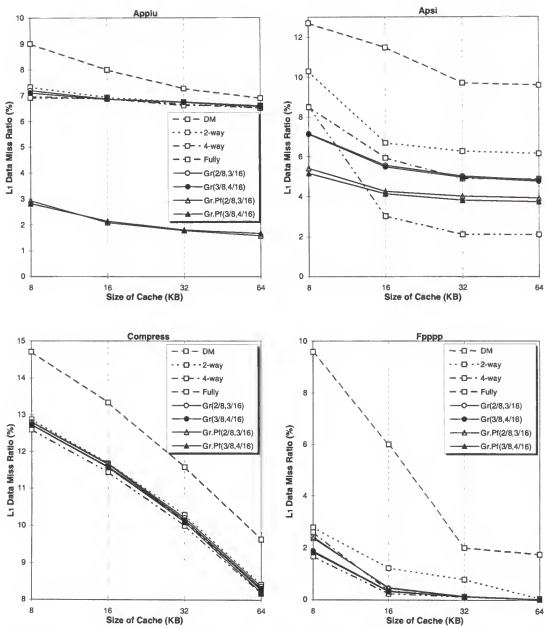


Figure 3.6. Cache miss ratio for workloads Applu, Apsi, Compress, and Fpppp

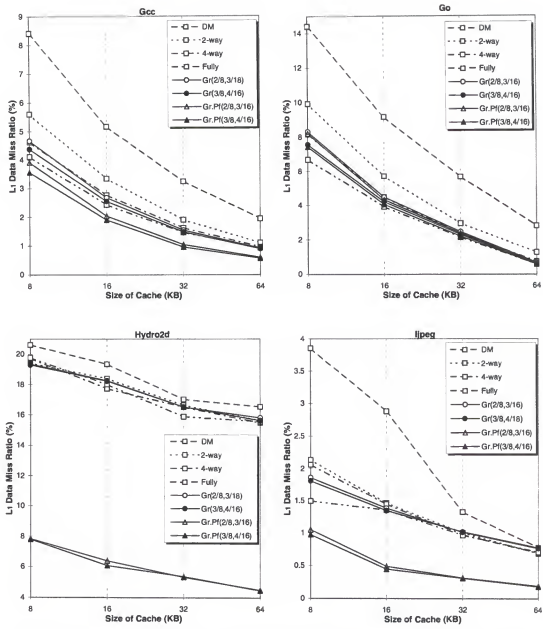


Figure 3.7. Cache miss ratio for workloads Gcc, Go, Hydro2d, and Ijpeg

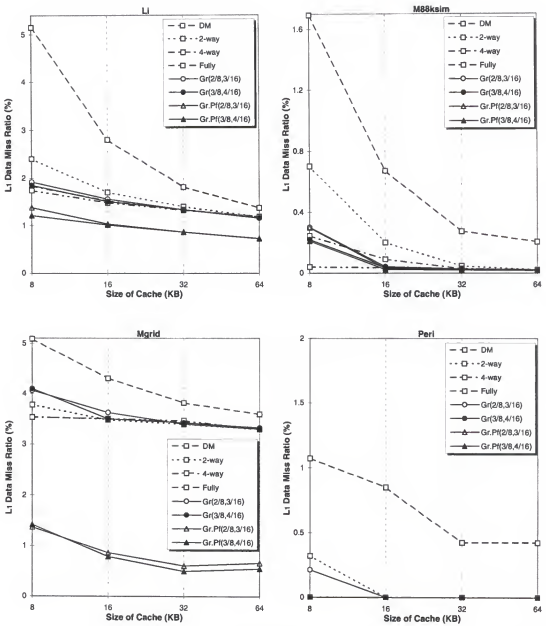


Figure 3.8. Cache miss ratio for workloads Li, M88ksim, Mgrid, and Perl

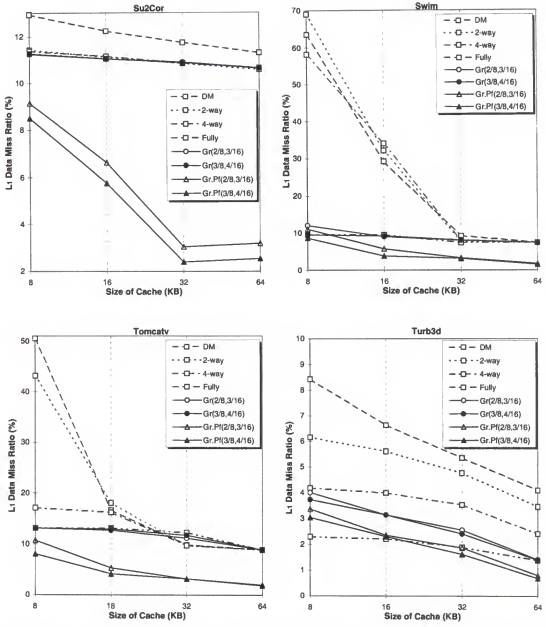


Figure 3.9. Cache miss ratio for workloads Su2cor, Swim, Tomcatv, and Turb3d

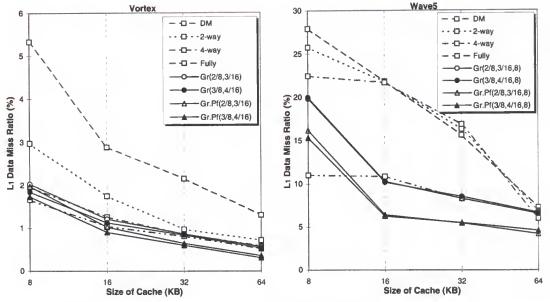


Figure 3.10. Cache miss ratio for workloads Vortex and Wave5

fact that memory references with a constant stride are very common in these programs. Such a reference pattern results in heavy conflicts and pollutes the entire fully-associative cache. On the other hand, the group-associative cache is able to handle the conflicts by effectively using the holes while confining the pollution to a subset of the cache.

The filtered sequential prefetch scheme improves the miss ratio of the group-associative cache for all the programs, especially those that are floating-point intensive. For instance, the respective improvement for Hydro2d and Su2cor are about $60-70$ % and $30-60$ %. For Su2cor, prefetching is especially effective for the larger caches. In addition, the difference between the two group-associative configurations is much bigger when prefetching is performed. This is because Su2cor has a large

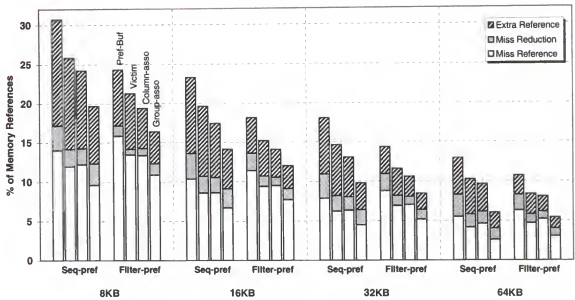


Figure 3.11. Miss ratio and memory traffic with data prefetching (TPC-C-like)

amount of stride references and an uneven reference distribution across the cache sets. In this case, a larger cache and a bigger SHT/OUT directory will enable the prefetched lines to be kept longer so that they are more likely to be referenced. Gcc, Li, Vortex, Tomcatv, Turb3d, and Wave5 also show sizable improvements in group-associative miss ratio with prefetching.

3.4.4 The Result of the Commercial Workload TPC-C-like

Figure 3.11 summarizes the miss ratio improvement as well as the increase in memory traffic that results from applying the two simple prefetch mechanisms to the various cache designs. Each column in the figure is divided into 3 segments. The bottom-most segment depicts the cache miss ratio with data prefetch. The second segment represents the miss ratio improvement that comes from prefetching

the data. The last segment reflects the net extra memory traffic that results when prefetching is performed. Note that this figure considers only the $(\frac{3}{8},\frac{4}{16})$ group-associative cache.

As expected, the two prefetching schemes both reduce the cache miss ratio markedly for all the cache designs. For instance, the 32 KB caches sequential prefetch reduces the miss ratio for the direct-mapped, victim, column-associative, and group-associative caches from 10.9 %, 8.2 %, 8.1 %, and 6.4 % to 7.9 %, 6.2 %, 6.3 %, and 4.4 % respectively. However, memory traffic is also increased considerably. Sequential prefetch increases memory traffic by 65 %, 78 %, 62 %, and 52 % respectively. The corresponding figures with filtering are 31 %, 41 %, 32 %, and 31 %.

A comparison of the direct-mapped cache with its separate prefetch buffer, the victim cache, and the column-associative cache shows the group-associative cache handles data prefetching better in terms of both the miss ratio improvement and the additional memory traffic. This is due to the fact that the group-associative cache is able to effectively control any cache pollution that may result from prefetching. In addition, the adaptive group associativity allows the prefetched lines to stay in the cache longer, thus increasing their chances of being used before replacement.

CHAPTER 4
DEFERRED CACHE COHERENCE MODELS

4.1 Introduction

The maturity of multiprocessing technology becomes an indispensable vehicle in building high-performance computer systems. One practical approach taken by recent microprocessors is to include the multiprocessor control function in the processor chip such that multiple microprocessors and memory modules can be connected by a system bus (also called a *snooping bus*) to form a Symmetric Multiprocessor (SMP) system. Each processor is typically equipped with a coherent cache memory to hide the memory access latency and to reduce the critical shared bus traffic. As the performance gap between processor and memory continues to widen, the demand for a higher bandwidth system bus and lower cache coherence traffic increases in order to sustain the memory requests from a moderate number of processors in SMP systems.

The key coherence function is to guarantee that each processor always observes all the memory writes from any other processor. Therefore, whenever a processor writes to a memory location, the old copy of the data existing in other caches must be updated or invalidated. Such a cache coherence activity not only incurs heavy overhead, but may increase cache miss ratios and system bus traffic

due to the false-sharing behavior. The false-sharing occurs when multiple processors attempt to update a different portion of a cache line roughly about the same time [6, 15, 16, 57]. In fact, this cache coherence activity can be deferred until the next synchronization instruction to allow multiple writers to update the same cache line as long as software maintains an order of reads and writes to each memory location [1, 14, 17]. Under the software model, proper synchronization instructions are inserted to enforce certain order of memory accesses from different processors. As a result, any data producer/consumer among the processors must be across a synchronization instruction.

This software synchronization model provides flexibility to the cache coherence protocol to enable writes by a processor to be visible to other processors. The traditional *eager* coherence protocol enforces the cache coherence on every memory write [39]. The *lazy* coherence protocol, on the other hand, permits temporary inconsistent cache copies [8, 9, 14, 25, 33, 35, 37, 51]. Those copies of data in different caches need to be reconciled only after the execution of a synchronization instruction. Although this lazy coherence protocol eliminates unnecessary cache coherence activities, it may still incur tremendous overhead on posting writes and reconciling cache lines at each synchronization instruction.

In this chapter, a *deferred cache coherence model* on bus-based SMPs is described, that further delays posting writes until a processor requests the new data. There are three fundamental techniques used in the proposed model. First, new partially modified states are added to the typical Modified-Exclusive-Shared-Invalid

(MESI) coherence protocol. The additional states allow multiple writers simultaneously updating different portions of the same cache line. Second, all the cache lines are *marked* at each synchronization instruction to indicate the need for reconciliation of certain inconsistent copies. In case the inconsistent cache lines are replaced before they are referenced, the reconciliation can be overlapped with normal instruction execution. Third, an efficient hardware *merging* technique is used to minimize the overhead associated with reconciliation. This technique uses the original copy of the data in memory to identify and merge the modified portion of a cache line for restoring the consistent copy. Cycle-by-cycle program-driven simulation of SPLASH-2 workload [61] show that the deferred coherence protocol can out-perform the eager protocol up to 30 %.

4.2 Statement of Problem

In order to minimize the traffic to the snooping bus, a number of cache coherence protocols have been proposed under two policies: *write-invalidate* and *write-broadcast* [5]. In both policies, read requests are carried out locally if a valid copy exists in the local cache. For write requests, the two policies work differently. When a processor updates a cache line, all other copies of the same cache line must be invalidated according to the write-invalidate policy to prevent other processors from accessing the stale data. Under write-broadcast policy, a write is broadcasted to all other caches to update the old copy, if a processor holds one of the copies. The

write-broadcast policy creates global traffic on every write and is not suitable for a bus-oriented multiprocessor system.

Under the write-invalidate protocol, a cache line update will cause invalidation of the line existing in other caches. Such an invalidation is necessary when the processor with the invalidated copy needs to access the updated data. However, if the processor only accesses the other portion of the cache line after invalidation, an unnecessary cache miss may be created. The terms, *true* and *false* sharing are referred to the above two conditions respectively.

Figure 4.1 illustrates simple examples of a true and a false sharing, when A and B are two distinct objects in the same cache line. It can be observed that the sharing behavior depends primarily on how the memory reference from multiple processors are interleaved. For instance, in the left example, *Read* A at t_2 happens after $Write$ A from P_1 thus a true-sharing miss occurs because P_2 wants the updated data from P_1. This sequence is reversed in the example on the right. The invalidation caused by $Write$ A of P_1 at t_2, incurs a false-sharing miss because P_2 does not need the updated A afterward. There have been a few attempts to precisely define the true and the false sharing behavior [24].

The semantics of the above two examples are totally different. In the first example, P_2 gets the newly updated A from P_1, while in the second example. P_2 reads the old value of A. In order to maintain proper order of write-read, read-write, and write-write sequences to the same data object, synchronization instructions

must be inserted as illustrated in the same Figure. Under this condition, the true-sharing can only happen across a synchronization point, and all the sharing between any two adjacent synchronization points belong to the false-sharing category.

Therefore, it is safe to cancel the invalidation due to cache line updates between synchronization points to eliminate false-sharing misses. However, the inconsistent cache line copies need to reconcile back to consistent state before it is accessed again after synchronization points.

Under the SPMD programming model, the synchronization variables are always kept in the coherence state to provide the latest update value. With the assistance of compiler, memory references can be partitioned into *coherence references* of the synchronization variables and the *coherence deferred reference* of the rest of the memory accesses. Also, the Critical Section (CS) which is protected by synchronization primitives are identified to indicate exclusive update to the global variables. In addition, hardware must identify the end of a barrier in order to indicate the points which prevent a processor from accessing cache lines which is inconsistent to the memory. This can be accomplished by marking a bit when a processor reaches a barrier.

4.3 Synchronization Primitives

In the popular Single-Program-Multiple-Data (SPMD) computation of parallel programs, the same program is sent to the participating processors and individual

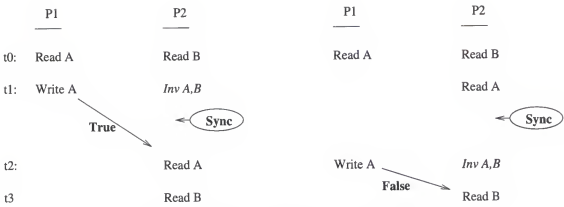


Figure 4.1. True and false sharing examples

processors execute different pieces of work by operating on different pieces of data. A parallel program is laid out as a sequence of code regions separated by *barrier* synchronization primitives. At each barrier, no process is allowed to proceed until all the processes participating in the barrier synchronization arrive this enforces proper data dependency. In addition to barriers, the other common synchronization requirement in SPMD is to allow mutual-exclusive updates of certain shared variables such as loop indices, processor IDs, etc. by the individual processors during the course of parallel execution.

The barrier and the mutual exclusion are the two synchronization primitives used in the parallelized SPLASH-2 application suites [61]. Both primitives can be implemented by using the basic *Lock* and *Unlock* instructions. A section of codes that are protected by the lock/unlock instructions can update any shared variable exclusively. Such a code section is called a *critical section.* A barrier can be implemented by counting the number of processes arriving at the barrier. The update of the barrier counter by each process must also be protected by a critical

```
barrier(int mutex) {
   lock(mutex);
   barrier_count++;
   unlock(mutex);
   while (barrier_count < total_count) ;
}
```

Figure 4.2. Barrier synchronization primitives

section. Figure 4.2 shows an example of implementing a barrier using lock/unlock. In this example, a synchronization variable *mutex* is used. This variable can be read-modified-written as an indivisible operation by hardware instructions such as test-and-set, compare-and-swap, etc. to accomplish the lock/unlock function. The critical section can be generalized to handle mutual-exclusive updates of one or more shared variable.

The use of mutual exclusion and barrier synchronization presents two fundamental properties. First, a barrier must be inserted to enforce a producer/consumer data dependence between two processes. As a result, the updates of a shared variable only needs to be observed by the consumer process after the execution of a barrier. Second, although the mutual-exclusive updates of a shared variable in a critical section can be executed in any order, the updates must be observed by other processes once the updating process leaves the critical section. Based on these observations, a *deferred* coherence protocol can be designed to minimize the unnecessary cache coherence activities on shared-bus SMPs.

4.4 Deferred Cache Coherence Protocol

Under the SPMD synchronization model, the observation of a write by other processors can be deferred until after the next synchronization barrier. In other words, between barriers, multiple writers can simultaneously update different portions of the same cache line. There are two critical performance issues in a multiple-writer coherence protocol. The first one is how and when each processor notifies the writes to other processors. The second issue is how and when the inconsistent data created by the multiple writers are reconciled and posted. The write notification and data reconciliation are separated so that the updated data is posted only when another processor asks for the data.

In the proposed deferred coherence protocol, two new *partially modified* states (P and Q) are added to the popular snooping-bus, write-invalidate, write-back, Modified-Exclusive-Shared-Invalid (MESI) protocol [20, 26, 38, 40, 57]. A line in the P-state means the line is valid and has been modified by the local processor, meanwhile, such a line may also be valid and possibly modified in other caches. A Q-state line is very similar to the P-state line except that the the line has not been modified by the local processor. The purpose of separating the Q-state from the P-state is to eliminate extra memory write-back during the cache line reconciliation.

Upon a write hit, the S-state is changed to the P-state, and the state is changed to Q for the same line existing in any other cache. When a write miss occurs, the new line is set to the P-state if the line is also valid in any other cache. In addition,

an M-state line is changed to the P-state when another processor encounters a write miss to the same line. In case of a read miss, the new line state is set to Q if the line is in the P-state in any other cache. In essence, the additional P and Q states allow a line to be shared and modified in multiple caches.

Since the lines in the P and Q states will be stale after each synchronization barrier, it is necessary to invalidate/reconcile those inconsistent lines. In order to alleviate overhead and bus congestion, the reconciliation of the P-state lines can be further deferred until the line is first accessed or replaced after the barrier. However, each processor must be notified of the potential stale cache lines existing in its own cache. In this regard, one extra bit called the *mark bit* is added to each entry in the cache tag directory. All the mark bits in every cache directory are *set* after each synchronization barrier to indicate the potential stale lines. Note that the mark bits can be implemented as a separate array for fast set/reset of all the bits. The corresponding mark bit is reset at the first access after a barrier to resume normal coherence activities to the cache line. The access of the marked line follows the rules listed below.

1. Marked P-state: The line must be reconciled upon the first access from either the local processor or the remote processor through the snooping bus. The request will be reissued after the reconciliation.
2. Marked Q-state: The line is invalid.
3. Marked M, E, S, and I states: The same as the corresponding unmarked states.

In addition to trigger the reconciliation once the marked P-state line is accessed, reconciliation of individual line is also required when both marked and unmarked P-state lines are replaced from the cache. The detailed deferred coherence protocol is given next followed by the description of the reconciliation in the following section.

Requests from local processor:

1. *Read hit:* When the requested line is found in the local cache, the data can be accessed and the state M, E, S, P, or Q remains unchanged.
2. *Write hit:* When the state is S, a global request will send to other processors and the state is change to P afterwards. Otherwise, the write can be performed locally. The states E and Q are changed to M and P respectively, while states P and M remain unchanged.
3. *Read miss:* A line-fill request is issued. The new state is E when the requested line is not present in any other cache. If the requested line is M in another cache, the cache which owns the modified copy supplies the data, and the new state is also set to E. If the requested line exists in other caches in any other state, E, S, or P, the new state becomes S, S, and Q respectively and the line is fetched from the memory.
4. *Write miss:* Write-allocation is assumed in this design. Upon a write-miss, the target line is fetched into the local cache. The memory supplies the data and the new state is M when the requested line does not exist in any other cache. Otherwise, the new state becomes P.
5. *Hit a marked P-state or replace a P- or marked P-state line:* Cache line reconciliation is initiated. A detailed description will be given in the next section.

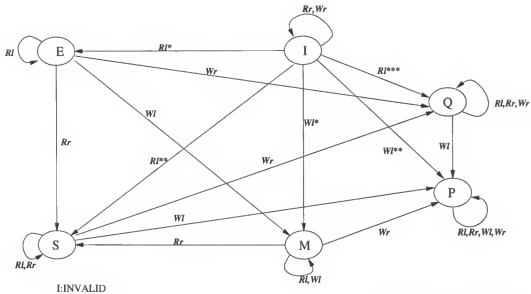
Requests from snooping bus:

1. *Snooping read hit:* Upon a snooping read hit, the processor sends the modified (M-state) data to the requester and the memory. The line becomes S afterwards. In case of a hit to an E-state, the state is changed to S without involving any data transfer. No action is taken when the state is either S, P, or Q.
2. *Snooping write hit:* When a snooping write hits a M-state, no data transfer is needed and the state is changed to P. If the line state is E or S, the line is changed to Q-state. No action is taken for the remaining states.

3. *Snooping write-back hit to a P- or marked P-state line:* Upon a hit, the snooping agent also writes back the P- or marked P-state line so that the reconciliation of the line can be carried out at the memory controller. The line is invalidated afterwards. A detailed description of the reconciliation will be given in the next section.

The state transition diagram of the deferred coherence protocol is given in Figure 4.3. In the figure, R represents a read request, W represents a Write request, and the subscript l and r represents local and remote requests respectively. Note that for simplicity, all the marked states are omitted since they are essentially unchanged except for the marked P and Q states. The marked P-state, once accessed either locally or remotely will trigger the line reconciliation. The marked Q-state line is invalid.

Note that all the memory references outside the critical section are followed by the deferred coherence protocol to maintain cache coherence. However, any update within the critical section need to be posted right after the lock is released. In addition, the update of the synchronization variable in lock/unlock must trigger an eager coherence request so that the other processors will not access any stale data. Since the critical section is generally very small in all the SPLASH-2 applications, the use of the eager MESI coherence protocol is reasonable for all the memory references in the critical section including lock/unlock.



I:INVALID
E:EXCLUSIVE
S:SHARED
M:MODIFIED
P:PARTIALLY-MODIFIED (Dirty)
Q:PARTIALLY-MODIFIED (Clean)

Rl : Read from Local Processor
Rr : Read from Remote Processor
Wl : Write from Local Processor
Wr : Write from Remote Processor

Rl*: No other cache has a copy of the requested line

Wl**: The requested line is fount in other cache

Figure 4.3. State transition of the Deferred Coherence Protocol

4.5 Reconcile Partially-Modified Cache Lines

At the end of a barrier, all the involved processors set the *mark* bits of all the cache entries to indicate the potential stale data. Proper synchronization is necessary to guarantee the completion of all the marks before any processor can proceed. Cache line reconciliation is necessary when a processor or a snooping request accesses a marked P-state line, or when a P-state or a marked P-state line is replaced from a cache.

When a partially modified line write-back is initiated, all other processors that have the same P-state or marked P-state line also write-back their own copy of the line. It is essential to provide an efficient way of merging those partially modified cache line copies to restore the consistent copy in the memory. To eliminate the need of recording the portion of the cache line being modified in each processor, a partially modified cache line *merging* technique is used to reconcile the inconsistent copies. This technique requires a special design of the memory controller to handle the merge. Upon receiving the partially modified write-back request, the memory controller begins to merge the original cache line content stored in the memory with the new copies of the cache line from the involved processors in a pipeline fashion. Basically, the merging algorithm performs a sequence of *exclusive-or* logic operations to identify the bit positions where the modification has been made. The new cache content can then be obtained by complementing the values of those bit

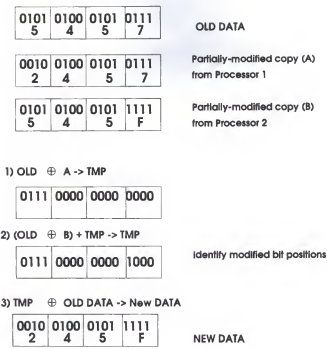


Figure 4.4. A reconciliation example

positions as indicated in the following logic equation, where n is the number of partially modified copies for the merge, and $\bigcup$ represents the logic OR operator.

$$New\ Data\ =\ Old\ Data\ \oplus\ (\ \bigcup_{i=1}^{n}(Old\ Data) \oplus (P - state\ Data)_i\ )$$

To simplify the memory controller design, it is assumed that the multiple copies of the partially modified line arrive at memory controller continuously without being intervened by other requests. In order to satisfy this requirement, the processor from which the P-state write-back is initiated must hold the data bus until the completion of all the write-backs. In addition, the snooping controllers, once receiving a P-state write-back request, will preempt any other request. A simple example which

illustrates the merging operation is also given in Figure 4.4. Since the P-state write-back involves multiple copies of a cache line and each copy must be sent sequentially through a shared data bus, the overhead associated with the merging operations can be overlapped with the P-state line transfer.

Note that the processor requesting a marked P-state line will reissue the request after the reconciliation. However, in case there is only a single copy in the marked P-state, the state is changed to E. The reissue of the request is not needed when the only marked P-state copy is located locally.

4.6 Performance Evaluation

Mint, a program-driven MIPS-based multiprocessor simulation tool [59, 60] was used for the dissertation research to compare the performance of the eager and the deferred coherence protocols under the SPLASH-2 workload [61]. Among 12 applications of SPLASH-2 workloads, Cholesky, FFT, LU, and Radix are computational kernel programs and Barnes, Fmm, Ocean_Cont., Ocean_Non., Raytrace, Volrend, Water_Nsq., and Water_Sp. are complete applications. The programs represent a variety of scientific, engineering, and graphics applications. The default input values are used for all the SPLASH-2 workloads.

4.6.1 Simulation Model

A simple processor model was developed. Each instruction takes one cycle to execute using a perfect branch predictor when the instruction is found in the L_1 instruction cache. The load/store instruction also takes a single cycle if both the instruction and the data are located in the L_1 caches. A delay of 4 cycles is charged when the instruction or the data is not located in the L_1 cache, but present in the L_2 cache. Further delays are incurred according to the detailed bus/memory cycles when the requested instruction and data miss the L_2 cache. In the simulation, a separate direct-mapped instruction and data L_1 cache of 8 Kilo-Bytes (KB) with a 512KB combined 4-way set-associative L_2 cache are used. The cache line size is 64 bytes for both L_1 and L_2 caches. Note that a write miss will not stop the processor pipeline until another miss is encountered or the write buffer is full.

Split-transaction snooping buses based on the MESI and the deferred coherence protocols are modeled. In view of the current technology, the processor cycle is four times faster than the bus cycle to reflect slow on-board system bus. The snooping bus consists of separate *command/address bus* (or simply *command bus*) and *data bus*. The width of the data bus is 16 bytes and a 64-byte cache line can be transferred in 4 consecutive cycles. The command bus can accept a request every three cycles. Once a request is active, it requires two cycles for bus arbitration. The command and address are issued right after the bus is granted. It then takes two cycles for each processor to look-up and update the cache directory for the snooping

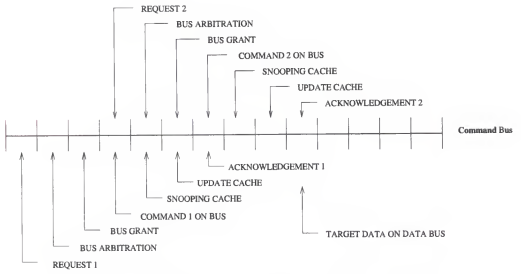


Figure 4.5. The split-transaction snooping bus design

request. Finally, an acknowledgment from each processor is sent to the requester as illustrated in Figure 4.5.

A bus request can be rejected when it issues a request to a line in the transient state, i.e., the line is in the process of satisfying the request from another processor. A negative acknowledgment may also be received when the requested line is in the marked P-state. The rejected request will be reissued later. To avoid excessive retry, the retry from 5 to 20 bus cycles are delayed based on a random number generator.

For a normal read miss, it takes 10 bus cycles to receive the target 16 bytes from the memory. Additional 6 bus cycles are required when the requested data is modified in another cache and a cache-to-cache line transfer is required due to the inquiry and bringing out the modified copy. For the P-state or marked P-state line write-back, it takes 7 bus cycles for the first processor including bus arbitration

Table 4.1. Cache hit/miss and reconciliation penalties

TRANSACTION	PROCESSOR CYCLE
Ideal instruction execution	1
Read L_1 miss	4
Read L_2 miss: data from memory	10×4
Read L_2 miss: cache-to-cache transfer	16×4
Write Miss	1 (up to 4×4)
P-state write-back: 1st processor	7×4
P-state write-back: others	4×4
Retry request	(5 to 20) $\times 4$

and data transfer. Afterwards, each of the rest processors can write the partially modified lines back to the memory in every 4 consecutive cycles. The processor can execute one instruction at every cycle unless a read miss or a cache reconciliation is required. The processor can handle one bus transaction at a time, and stalls on a read miss until the data comes back. A 2-entry write buffer is assumed in each processor. Upon a write miss, the processor continues execution until the write buffer is full or a read miss is encountered. Table 4.1 summarizes the penalty cycles.

In addition to the eager MESI protocol, the deferred coherence protocol is also compared with a similar delay coherence protocol [14]. Instead of using the merging technique based on the content of the partially modified cache lines, the delay coherence protocol implements a small Invalidation Send Buffer (ISB) to delay the write posting and to remember the modified portion of the cache line. In this design, the overflow of the ISB may create excessive write traffic. Meanwhile, the

reconciliation of the partially modified line at each synchronization instruction may incur extra overhead compared with the deferred lazy coherence protocol. In the simulation, each processor has four-entry ISB.

4.6.2 Performance Comparison

The results of the total execution time of all the applications of SPLASH-2 on 32 and 64 processor systems are plotted in Figure 4.6. The total execution time is the average execution time of all the processors. Note that in this figure, the execution time is normalized with respect to the execution time under the MESI coherence protocol on a 32-processor system. The total execution time is divided into 7 timing components. The ideal cycle is the time to execute the respective program without any stall, i.e., the CPI is equal to 1. The read/write misses represent the delay associated with read and write penalties. Note that the write miss penalty includes the penalty associated with writes to S-state lines. Next, the stalls due to barriers and critical sections are charged separately. Finally, the write-back delay of the modified copy when the ISB is full and the merge/reconciliation delay in the deferred scheme is also accounted for separately.

Several important observations can be made from the Figures. First of all, the ideal execution times of the applications are almost reduced to half when the number of processors increases from 32 to 64. However, the penalty cycles are considerably larger on 64 processors than on 32 processors. In fact, in some cases,

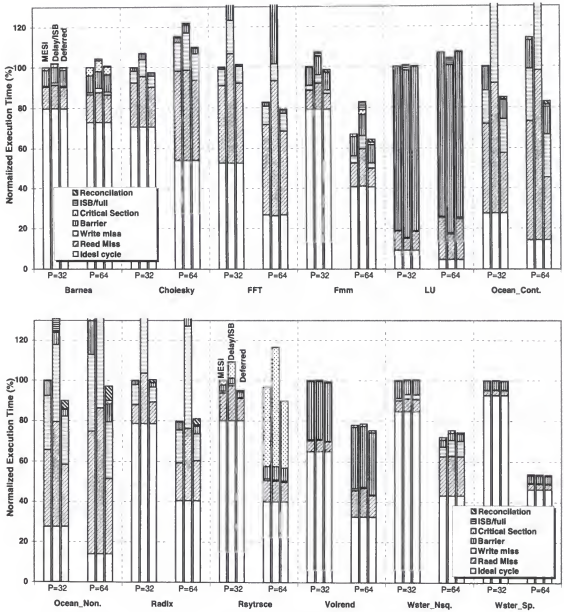


Figure 4.6. Normalized execution time of SPLASH-2 applications

such as Cholesky, the total execution time increases on the 64-processor systems for all three coherence schemes. The primary reason is due to bus congestion that makes the read/miss penalties, as well as the delay of write-backs and reconciliation heavier on the 64-processor systems. As can be seen from Figure 4.7, when both the command and data bus utilization are beyond 60%, increased penalty cycles can be observed from several timing components in Figure 4.6.

Generally speaking, the deferred coherence scheme shows better performance than that of the MESI and the delay protocols. For the 64-processor systems, the deferred scheme shows about 6%, 4%, 3%, 30%, 30%, 7%, and 3% improvement of the total execution time over the conventional MESI protocol for the respective Cholesky, FFT, Fmm, Ocean_Cont., Ocean_Non., Raytrace, and Volrend. This is mainly due to the fact of significant reduction in read/write penalties. For the workloads, Barnes, LU, and Water, the performance of the Deferred coherence model is similar to the MESI coherence model. The workload, Radix, is the only workload which the Deferred coherence model performs worse than MESI coherence model, however, the difference is about 1%. Figure 4.8 shows the L_2 miss ratios of the three coherence schemes for all applications. Both the delay and the deferred schemes improve the miss ratios compared with the MESI protocol. This is because both schemes allow shared and modified copies in multiple caches. The deferred scheme can further improve the miss ratio for Ocean_Cont., Ocean_Non., Raytrace, and Fmm, because the deferred protocol changes the marked P-state to E-state upon reconciliation when no other cache holds the the same line in the marked P-state.

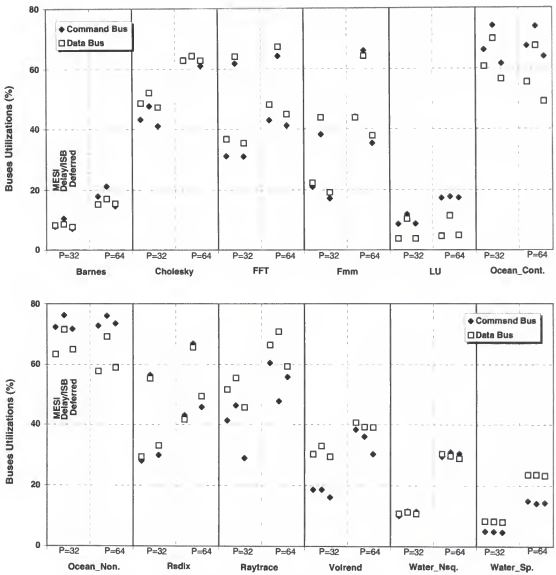


Figure 4.7. Command and data bus utilization

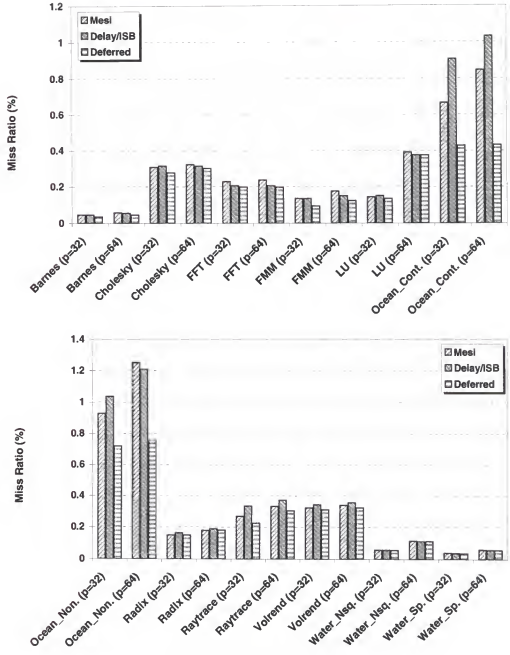


Figure 4.8. L_2 miss ratios

The reduction of the number of misses for the delay and the deferred schemes can be seen easily by counting the number of bus requests as shown in Figure 4.9. In this figure, the number of shared bus requests are plotted based on several categories: read miss, write miss, cache-to-cache transfer, write to S-state, write-back, reconciliation, and retried request due to negative acknowledgment. A request can be rejected if it accesses a line in the transient state, i.e., the line is in the process of satisfying the request from another processor. The negative acknowledgment may also be received when the requested line is in the marked P-state. A significant reduction in term of the number of read/write misses to the shared bus can be observed for the deferred scheme especially when running under Barnes, Ocean_Cont., Ocean_Non., Raytrace, and Fmm. In comparison with the results in Figure 4.6, however, the reduction of the miss penalty may not be as significant. This is due to the fact that cache reconciliation occupies the data bus and may affect other regular misses or write-backs.

Another interesting observation with regard to the penalty and the frequency of the reconciliation can be made from Figure 4.6 and Figure 4.9. Figure 4.9 indicates a noticeable amount of reconciliation requests for Barnes, FFT, Fmm, and Ocean_Cont., Ocean_Non., Radix, and Water_Sp. In contrast, only Ocean_Cont., and Ocean_Non. shows noticeable impact to the total execution time by the reconciliation. This is due to the fact that a majority of the reconciliations are triggered by the replacement of the P- and marked P-state lines in Barnes, FFT, Fmm, Radix,

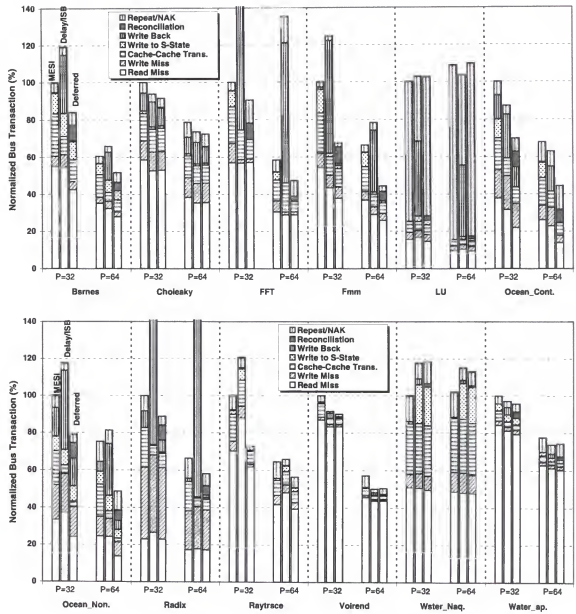


Figure 4.9. Normalized bus transactions

and Water_Sp. The processor can continue normal execution in parallel with the replacement reconciliation.

The delay protocol has the worst performance for most applications even though the L_2 miss ratio is actually improved from that of the MESI protocol. Given the fact that the delay protocol relies on the small ISB to remember recently modified words, the overflow of the ISB may generate an excessive write-back traffic (Figure 4.9). Such traffic not only incurs overhead, but also affects other bus requests. For instance, for 64 processors the miss ratio of FFT is reduced from .25% using the MESI protocol to about .21% using the delay protocol. But, the normalized execution time for the read/write miss penalties is increased from 55% to 75%. To verify the impact of ISB size, 16-entry ISB model were also simulated. In the worst case of FFT and Fmm, the improvement in total execution time is only about 20% and 2% respectively. Besides the ISB overflow, the delay protocol writes back all the modified copies from ISB at each synchronization barrier that incurs additional delay too.

4.7 Related Work

Dubois et al. introduced send-and-received delayed protocol which allows multiple copies of cache line in the "stale" state and keeps track of partially written words in the Invalidation Send Buffer (ISB) [14]. When the cache line is partially written, the updated word is stored in ISB. Whenever a synchronization primitive

is encountered, the stored information in the ISB is written to main memory for consistency. In other word, this scheme posts all the writes at each synchronization instruction. In addition, the overflow of the ISB may produce excessive write-back traffic.

The *Munin* [8] employs multiple consistency protocols annotating each shared variable declaration by expected access pattern. Munin, then chooses a consistency protocol suited to that pattern. Incorrect declaration should be detected by Munin's run time system. Munin is a software DSM system which provides a release-consistency memory interface. Pending outgoing writes are buffered and merged in a delayed update queue (DUQ), the DUQ is flushed whenever a local thread releases a lock or arrives at a barrier. The modified copies are propagated and merged with their remote copies.

Knotothanassis et al. introduced a hardware cache coherence protocol [35] which maintains multiple writes (the multiple write state is called "Weak" state), and the protocol needs the process of updating a processor's weak list. The number of weak list should be limited to avoid large write backs at the time of synchronization. Anderson and Baer used the sector cache in their a coherence protocol to reduce false sharing misses [4]. The basic idea of this coherence protocol is that it favors cache to cache transfer, and the transferred *dirty* sub-blocks on read miss are not copied into memory. However, in normal cache coherence under bus-based multiprocessor, the cache-to-cache transfer is expensive transaction.

Kaxiras et al. described the GLOW cache coherence protocol [32], which is the extension to Scalable Coherence Interface(SCI) of ANSI/IEEE standard. The GROW cache coherence protocol handles coherence activities for the widely shared data. If a request is the reference request to widely shared data, the special hardware called the GLOW agent intercepts the request and handles the request differently. The cache coherence protocol is either invalidation or update for shared write.

Iftode et al. proposed the Automatic Update Release Consistency (AURC), the AURC is based on lazy consistency model [25]. By snooping on the memory bus, the automated update mechanism propagates update to shared data at a word granularity as long as the mapping has been established between local page and remote page to which the write propagates. The mappings are established as soon as more than one processor accesses the shared page. The basic approach is to have *home* memory for every shared page and to set up mapping such that writes to other copies of the page are automatically propagated to the home.

CHAPTER 5
CONCLUSION

Cache memory performance plays an important role to the computer system performance. For the single processor system, improving the hit ratio without degrading cache access time can achieve a fast average memory access time. Also, for the multiple processor system, reducing requests to the snooping bus can decrease the execution time for the parallel programs.

In this dissertation, it is observed that the direct-mapped cache, instead of faithfully maintaining the lines that have been referenced recently, retains a large number of less-recently used lines that are not likely to be re-referenced before they are replaced. Based on this observation, an adaptive group-associative cache is proposed. This cache is able to dynamically identify the underutilized cache frames and to effectively use them to selectively retain some of the lines that are to be replaced. Performance evaluation using trace-driven simulations of both the SPEC95 benchmark suites and TPC-C-like benchmark show that the group-associative cache is able to decisively outperform the conventional and various performance-enhanced cache organizations. In particular, the miss ratio of the adaptive group-associative cache is consistently better than that of the 4-way set-associative cache and, in some cases, even approaches that of the fully-associative cache. As a result, the

adaptive group-associative cache has the lowest average memory access time among the different cache organizations.

Furthermore, the group-associative cache is able to handle data prefetching better than other cache organizations. In terms of cost, a first-cut estimate shows that the directories of the group-associative cache require about 4.5% of the area taken up by the cache.

The deferred cache coherence model with a new partially-modified state is designed and evaluated. The unique feature is to allow inconsistent copies of a modified cache line in multiple cache temporarily to circumvent the adverse effect of the false-sharing behavior in parallel programs. Program-driven, cycle-by-cycle simulation of a snooping-bus multiprocessor models are developed to evaluate the performance of the proposed method. The results based on the parallel applications from SPLASH-2 show that the proposed cache coherence protocol can improve the performance over the conventional MESI scheme and delayed cache coherence protocol.

In summary, the group-associative cache and the deferred cache coherence model are proposed ways of improving the cache memory performance for the single processor system and multiple processor system. The simulation results using SPEC95 workloads, TPC-C-like workload, and SPLASH-2 workloads show that the proposed designs improve the cache memory performance.

APPENDIX A
Compulsory, Capacity, and Conflict Misses for SPECint95

In order to understand the compulsory, capacity and conflict misses with different cache topology and cache size. Various cache topologies, which are direct-mapped cache, 2-way set-associative cache, 4-way set-associative cache and fully-associative cache, are simulated using SPEC95 benchmark programs.

In the tables, "SZ" means "size of cache," "ASSO" means "associativity," "COM" means "Compulsory misses," "CAP" represents "Capacity misses," and "CON" means "Conflict misses." Also, "ACT" represents the "Actual misses" and "REL" means the "Relative percentage."

Table A.1. Compulsory, capacity, and conflict misses of Compress

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	14.7	0.20	1.38	12.4	84.3	2.11	14.4
		2-way	12.9	0.20	1.58	12.4	96.3	0.28	2.16
		4-way	12.7	0.20	1.59	12.4	97.3	0.15	1.14
	16	D.M.	13.3	0.20	1.52	11.2	84.2	1.90	14.3
		2-way	11.7	0.20	1.74	11.2	96.2	0.25	2.10
		4-way	11.6	0.20	1.75	11.2	97.1	0.13	1.15
	32	D.M.	11.6	0.20	1.75	9.79	84.5	1.59	13.7
		2-way	10.3	0.20	1.97	9.79	95.2	0.29	2.82
		4-way	10.1	0.20	2.01	9.79	96.8	0.12	1.21
	64	D.M.	9.63	0.20	2.11	7.97	82.7	1.46	15.1
		2-way	8.42	0.20	2.41	7.97	94.7	0.24	2.90
		4-way	8.29	0.20	2.45	7.97	96.2	0.11	1.35
Inst Cache	8	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	16	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	32	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	64	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA

Table A.2. Compulsory, capacity, and conflict misses of Gcc

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	8.41	0.01	0.157	4.10	48.8	4.30	51.1
		2-way	5.60	0.01	0.24	4.10	73.3	1.48	26.4
		4-way	4.63	.013	.29	4.10	88.7	0.51	11.0
	16	D.M.	5.17	0.01	0.26	2.41	46.7	2.74	53.1
		2-way	3.34	0.01	0.40	2.41	72.1	0.92	27.5
		4-way	2.77	0.01	0.48	2.41	87.2	0.34	12.3
	32	D.M.	3.25	0.01	0.41	1.45	44.7	1.78	54.9
		2-way	1.90	0.01	0.69	1.45	76.2	0.44	23.1
		4-way	1.62	0.01	0.82	1.45	89.5	0.16	9.69
	64	D.M.	1.95	0.01	0.68	0.92	47.1	1.02	52.2
		2-way	1.11	0.01	1.20	0.92	83.0	0.18	15.8
		4-way	0.97	0.01	1.37	0.92	94.8	0.04	3.83
Inst Cache	8	D.M.	5.77	0.002	0.03	5.21	90.3	0.56	9.66
		2-way	5.36	0.002	0.03	5.21	97.2	0.15	2.77
		4-way	5.24	0.002	0.03	5.21	99.6	0.02	0.42
	16	D.M.	3.81	0.002	0.04	2.33	61.0	1.49	39.0
		2-way	3.15	0.002	0.05	2.33	73.9	0.82	26.0
		4-way	2.80	0.002	0.06	2.33	83.1	0.47	16.8
	32	D.M.	2.01	0.002	0.08	0.59	29.4	1.42	70.5
		2-way	1.38	0.002	0.12	0.59	42.9	0.79	57.0
		4-way	1.0	0.002	0.16	0.59	59.8	0.4	40.1
	64	D.M.	1.05	0.002	0.15	0.19	18.1	0.86	81.8
		2-way	0.45	0.002	0.36	0.19	42.0	0.26	57.7
		4-way	0.29	0.002	0.56	0.19	66.3	0.09	33.1

Table A.3. Compulsory, capacity, and conflict misses of Go

	SZ	ASSO	MISS	COM.MISS%		CAP.MISS%		CON.MISS%	
	KB		%	ACT.	REL.	ACT.	REL.	ACT.	REL.
Data Cache	8	D.M.	14.4	.001	0.01	6.68	46.4	7.73	53.6
		2-way	9.93	.001	0.01	6.68	67.3	3.24	32.7
		4-way	8.18	.001	0.01	6.68	81.7	1.49	18.2
	16	D.M.	9.16	.001	0.01	3.9	42.6	5.25	57.4
		2-way	5.67	.001	0.02	3.9	68.5	1.80	31.5
		4-way	4.47	.001	0.02	3.9	87.3	0.56	12.6
	32	D.M.	5.67	.001	0.02	2.13	37.6	3.54	62.4
		2-way	2.94	.001	0.04	2.13	72.5	0.81	27.5
		4-way	2.36	.001	0.05	2.13	90.2	0.23	9.74
	64	D.M.	2.80	.001	0.04	0.56	20.0	2.24	80.0
		2-way	1.27	.001	0.09	0.56	44.1	0.71	55.8
		4-way	0.91	.001	0.12	0.56	61.5	0.35	38.3
Inst Cache	8	D.M.	3.59	0	NA	3.03	84.4	0.56	15.59
		2-way	3.17	0	NA	3.03	95.6	0.14	4.41
		4-way	2.98	0	NA	3.03	101	-0.1	-1.0
	16	D.M.	2.63	0	NA	1.12	42.6	1.51	57.4
		2-way	1.44	0	NA	1.12	77.8	0.32	22.2
		4-way	1.27	0	NA	1.12	88.2	0.15	11.8
	32	D.M.	1.60	0	NA	0.39	24.4	1.21	75.6
		2-way	0.94	0	NA	0.39	41.5	0.55	58.5
		4-way	0.43	0	NA	0.39	90.7	0.04	9.23
	64	D.M.	0.72	0	NA	0.35	48.6	0.37	51.3
		2-way	0.48	0	NA	0.35	72.9	0.13	27.0
		4-way	0.39	0	NA	0.35	89.7	0.04	10.2

Table A.4. Compulsory, capacity, and conflict misses of Ijpeg

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	3.86	0.05	1.19	1.45	37.6	2.36	61.2
		2-way	2.13	0.05	2.14	1.45	67.9	0.64	29.9
		4-way	2.06	0.05	2.22	1.45	70.4	0.57	27.4
	16	D.M.	2.88	0.05	1.59	1.31	45.5	1.53	52.9
		2-way	1.47	0.05	3.12	1.31	89.3	0.11	7.57
		4-way	1.45	0.05	3.16	1.31	90.6	0.09	6.22
	32	D.M.	1.33	0.05	3.45	0.92	69.4	0.36	27.2
		2-way	1.0	0.05	4.55	0.92	91.6	0.04	3.88
		4-way	0.98	0.05	4.69	0.92	94.3	0.01	1.03
	64	D.M.	0.77	0.05	5.91	0.66	85.4	0.07	8.67
		2-way	0.69	0.05	6.65	0.66	93.4	0	NA
		4-way	0.69	0.05	6.65	0.66	93.4	0	NA
Inst Cache	8	D.M.	0.06	0	NA	.001	2.05	0.06	97.9
		2-way	.004	0	NA	.001	29.3	.003	70.7
		4-way	.002	0	NA	.001	50.9	.001	49.1
	16	D.M.	.003	0	NA	.001	37.8	.002	62.2
		2-way	.002	0	NA	.001	62.9	.001	37.1
		4-way	.001	0	NA	.001	100	0	NA
	32	D.M.	.002	0	NA	.001	53.1	.001	46.9
		2-way	.001	0	NA	.001	100	0	NA
		4-way	.001	0	NA	.001	100	0	NA
	64	D.M.	.001	0	NA	0	NA	.001	100
		2-way	.001	0	NA	0	NA	.001	100
		4-way	.001	0	NA	.001	100	0	NA

Table A.5. Compulsory, capacity, and conflict misses of Li

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	5.15	.001	0.02	1.74	33.8	3.41	66.2
		2-way	2.4	.001	0.04	1.74	72.5	0.66	27.5
		4-way	1.84	.001	0.05	1.74	94.8	0.09	5.13
	16	D.M.	2.8	.001	0.03	1.48	52.8	1.32	47.2
		2-way	1.7	.001	0.05	1.48	86.9	0.22	13.1
		4-way	1.52	.001	0.06	1.48	97.4	0.04	2.58
	32	D.M.	1.81	.001	0.05	1.32	73.0	0.49	27.0
		2-way	1.4	.001	0.06	1.32	94.8	0.07	5.17
		4-way	1.34	.001	0.07	1.32	99.0	0.01	0.9
	64	D.M.	1.37	.001	0.07	1.18	86.3	0.19	13.7
		2-way	1.18	.001	0.08	1.18	99.7	.002	0.18
		4-way	1.18	.001	0.08	1.18	99.9	0	NA
Inst Cache	8	D.M.	3.11	0	NA	0.07	2.19	3.04	97.8
		2-way	1.15	0	NA	0.07	5.91	1.08	94.1
		4-way	0.49	0	NA	0.07	13.9	0.42	86.1
	16	D.M.	1.73	0	NA	0	NA	1.73	100
		2-way	0.41	0	NA	0	NA	0.42	100
		4-way	0.05	0	NA	0	NA	0.05	100
	32	D.M.	1.01	0	NA	0	NA	1.01	100
		2-way	0.17	0	NA	0	NA	0.17	100
		4-way	.003	0	NA	0	NA	.003	100
	64	D.M.	0.76	0	NA	0	NA	0.76	100
		2-way	.001	0	NA	0	NA	.001	100
		4-way	0	0	NA	0	NA	0	NA

Table A.6. Compulsory, capacity, and conflict misses of M88ksim

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	1.69	0.01	0.51	0.03	1.85	1.65	97.6
		2-way	0.70	0.01	1.24	0.03	4.45	0.66	94.3
		4-way	0.24	0.01	3.57	0.03	12.8	0.20	83.6
	16	D.M.	0.67	0.01	1.29	0.03	3.9	0.64	94.8
		2-way	0.2	0.01	4.4	0.03	13.2	0.17	82.5
		4-way	0.09	0.01	9.67	0.03	29.2	0.06	61.1
	32	D.M.	0.27	0.01	3.18	0.02	7.04	0.25	89.8
		2-way	0.05	0.01	18.9	0.02	42.0	0.02	39.1
		4-way	0.03	0.01	30.0	0.02	66.6	.001	3.45
	64	D.M.	0.21	0.01	4.24	0.01	6.0	0.18	89.8
		2-way	0.02	0.01	41.4	0.01	58.6	0	NA
		4-way	0.02	0.01	39.5	0.01	61.5	0	NA
Inst Cache	8	D.M.	3.24	0	NA	1.05	32.4	2.19	67.6
		2-way	2.22	0	NA	1.05	47.3	1.17	52.7
		4-way	1.96	0	NA	1.05	53.7	0.9	46.3
	16	D.M.	1.3	0	NA	0.14	10.4	1.18	89.6
		2-way	0.63	0	NA	0.14	21.6	0.49	78.4
		4-way	0.29	0	NA	0.14	46.7	0.16	53.3
	32	D.M.	0.89	0	NA	0	NA	0.89	100
		2-way	0.12	0	NA	0	NA	0.12	100
		4-way	0.03	0	NA	0	NA	0.03	100
	64	D.M.	0.13	0	NA	0	NA	0.13	100
		2-way	0.06	0	NA	0	NA	0.06	100
		4-way	0	0	NA	0	NA	0	NA

Table A.7. Compulsory, capacity, and conflict misses of Perl

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	1.7	0	NA	0	NA	1.7	100
		2-way	0.32	0	NA	0	NA	0.32	100
		4-way	0	0	NA	0	NA	0	NA
	16	D.M.	0.85	0	NA	0	NA	0.85	100
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	32	D.M.	0.42	0	NA	0	NA	0.42	100
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	64	D.M.	0.42	0	NA	0	NA	0.42	100
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
Inst Cache	8	D.M.	3.89	0	NA	0	NA	3.89	100
		2-way	1.54	0	NA	0	NA	1.54	100
		4-way	0.62	0	NA	0	NA	0.62	100
	16	D.M.	1.33	0	NA	.001	0.08	1.33	99.9
		2-way	.002	0	NA	.001	50	.001	50
		4-way	.001	0	NA	.001	100	0	NA
	32	D.M.	0.51	0	NA	0	0.01	0.51	99.9
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	64	D.M.	0.41	0	NA	0	NA	0.41	100
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA

Table A.8. Compulsory, capacity, and conflict misses of Vortex

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	5.34	0.06	1.1	1.61	30.1	3.67	68.8
		2-way	2.97	0.06	2.0	1.67	54.1	1.31	43.9
		4-way	1.96	0.06	3.0	1.67	81.9	0.3	15.1
	16	D.M.	2.89	0.06	2.0	0.98	34.1	1.85	63.9
		2-way	1.75	0.06	3.3	0.98	56.2	0.71	40.5
		4-way	1.26	0.06	4.6	0.98	77.8	0.22	17.6
	32	D.M.	2.16	0.06	2.7	0.75	34.8	1.35	62.5
		2-way	0.98	0.06	6.0	0.75	77.0	0.17	17.0
		4-way	0.86	0.06	6.8	0.75	87.6	0.05	5.6
	64	D.M.	1.31	0.06	4.5	0.46	35.1	0.8	60.4
		2-way	0.72	0.06	8.1	0.46	63.7	0.20	28.2
		4-way	0.59	0.06	10.0	0.46	78.5	0.07	11.5
Inst. Cache	8	D.M.	3.98	0	NA	1.34	33.8	2.63	66.2
		2-way	3.32	0	NA	1.34	40.5	1.98	59.5
		4-way	2.0	0	NA	1.34	67.4	0.65	32.6
	16	D.M.	1.47	0	NA	0.65	44.3	0.82	55.7
		2-way	0.9	0	NA	0.65	72.8	0.24	27.2
		4-way	0.72	0	NA	0.65	90.2	0.07	9.8
	32	D.M.	0.89	0	NA	0.18	20.5	0.71	79.5
		2-way	0.38	0	NA	0.18	48.1	0.2	51.9
		4-way	0.22	0	NA	0.18	81.7	0.04	18.3
	64	D.M.	0.45	0	NA	0.01	2.8	0.44	97.2
		2-way	0.14	0	NA	0.01	9.2	0.13	90.6
		4-way	0.07	0	NA	0.01	19.5	0.05	80.0

APPENDIX B
Compulsory, Capacity, and Conflict Misses for SPECfp95

Table B.1. Compulsory, capacity, and conflict misses of Applu

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	9.01	0.14	1.56	6.78	75.3	2.09	23.2
		2-way	7.33	0.14	1.92	6.78	92.4	0.41	5.65
		4-way	6.97	0.14	2.02	6.78	97.3	0.05	0.69
	16	D.M.	8.0	0.14	1.76	6.74	84.3	1.12	14.0
		2-way	6.94	0.14	2.03	6.74	97.2	0.05	0.74
		4-way	6.88	0.14	2.05	6.74	98.0	0	NA
	32	D.M.	7.28	0.14	1.93	6.48	89.0	0.66	9.1
		2-way	6.76	0.14	2.09	6.48	95.9	0.13	1.97
		4-way	6.66	0.14	2.11	6.48	97.2	0.04	0.6
	64	D.M.	6.9	0.14	2.04	6.47	93.8	0.28	4.12
		2-way	6.5	0.14	2.16	6.47	99.6	-0.1	-1.7
		4-way	6.52	0.14	2.16	6.47	99.3	-0.1	-1.4
Inst Cache	8	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	16	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	32	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	64	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA

Table B.2. Compulsory, capacity, and conflict misses of Apsi

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	12.7	0.01	0.06	8.49	66.9	4.20	33.1
		2-way	10.3	0.01	0.08	8.49	82.4	1.80	17.5
		4-way	8.51	0.01	0.09	8.49	99.8	0.01	0.13
	16	D.M.	11.5	0.01	0.07	3.30	28.7	8.19	71.3
		2-way	6.68	0.01	0.12	3.30	49.3	3.38	50.6
		4-way	5.93	0.01	0.13	3.30	55.6	2.63	44.3
	32	D.M.	9.72	0.01	0.08	2.12	21.8	7.59	78.1
		2-way	6.26	0.01	0.12	2.12	33.8	4.14	66.1
		4-way	4.90	0.01	0.16	2.12	43.2	2.77	56.6
	64	D.M.	9.60	0.01	0.08	2.11	22.0	7.48	77.9
		2-way	6.15	0.01	0.13	2.11	34.35	4.03	65.5
		4-way	4.88	0.01	0.16	2.11	43.3	2.76	56.5
Inst Cache	8	D.M.	0.32	.0001	0.03	0.01	2.82	0.31	97.2
		2-way	0.13	.0001	0.08	0.01	7.22	0.12	92.7
		4-way	0.05	.0001	0.19	.009	17.3	0.04	82.5
	16	D.M.	0.10	.0001	0.10	0.01	4.61	0.10	95.3
		2-way	0.02	.0001	0.49	0.01	22.45	0.02	77.1
		4-way	0.01	.0001	1.82	0.01	83.8	.001	14.3
	32	D.M.	0.05	.0001	0.19	.004	7.05	0.05	92.8
		2-way	.004	.0001	2.67	.004	97.3	0	NA
		4-way	.003	.0001	2.92	.004	97.1	0	NA
	64	D.M.	.011	.0001	0.94	0	NA	.011	99.1
		2-way	.001	.0001	8.77	0	NA	.001	91.2
		4-way	.001	.0001	11.9	0	NA	.001	89.1

Table B.3. Compulsory, capacity, and conflict misses of Fpppp

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	6.50	0	NA	1.68	25.9	4.82	74.1
		2-way	2.81	0	NA	1.68	59.9	1.13	40.1
		4-way	2.45	0	NA	1.68	68.6	0.77	31.4
	16	D.M.	2.69	0	NA	0.34	12.8	2.34	87.2
		2-way	0.72	0	NA	0.34	47.8	0.38	52.2
		4-way	0.35	0	NA	0.34	99.7	.001	0.23
	32	D.M.	1.94	0	NA	0.12	6.03	1.82	94.0
		2-way	0.27	0	NA	0.12	43.2	0.15	56.8
		4-way	0.14	0	NA	0.12	91.4	0.01	8.44
	64	D.M.	1.73	0	NA	0.03	1.45	1.70	98.5
		2-way	0.06	0	NA	0.03	41.7	0.04	58.0
		4-way	0.04	0	NA	0.03	69.4	0.01	30.0
Inst Cache	8	D.M.	7.89	0	NA	8.89	112	-1.0	-12.7
		2-way	7.96	0	NA	8.89	111	-0.93	-11.6
		4-way	8.28	0	NA	8.89	107	-0.61	-7.4
	16	D.M.	6.53	0	NA	6.64	101	-0.1	-1.8
		2-way	6.57	0	NA	6.64	101	-0.1	-1.1
		4-way	6.65	0	NA	6.64	99.9	0.01	0.13
	32	D.M.	3.52	0	NA	2.06	58.5	1.46	41.5
		2-way	0.55	0	NA	2.06	375	-1.5	-275
		4-way	0.82	0	NA	2.06	252	-1.2	-152
	64	D.M.	0.03	0	NA	0	NA	0.03	99.7
		2-way	0.01	0	NA	0	NA	0.01	99.3
		4-way	0	0	NA	0	NA	0	NA

Table B.4. Compulsory, capacity, and conflict misses of Hydro2d

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	20.6	0.05	0.23	19.7	95.6	0.85	4.13
		2-way	19.4	0.05	0.24	19.7	101	-0.40	-1.8
		4-way	19.7	0.05	0.24	19.7	100	0	NA
	16	D.M.	19.3	0.05	0.25	17.9	92.8	1.35	6.97
		2-way	18.4	0.05	0.26	17.9	97.6	0.39	2.14
		4-way	17.9	0.05	0.27	17.9	99.7	0	NA
	32	D.M.	17.0	0.05	0.28	15.8	93.1	1.14	6.67
		2-way	16.6	0.05	0.29	15.8	95.2	0.75	4.51
		4-way	16.5	0.05	0.29	15.8	95.9	0.62	3.77
	64	D.M.	16.5	0.05	0.29	15.5	93.9	0.97	5.86
		2-way	15.6	0.05	0.30	15.5	99.3	0.07	0.44
		4-way	15.6	0.05	0.30	15.5	99.4	0.04	0.26
Inst Cache	8	D.M.	.002	0	NA	.002	100	0	NA
		2-way	.002	0	NA	.002	100	0	NA
		4-way	.002	0	NA	.002	100	0	NA
	16	D.M.	.001	0	NA	.001	100	0	NA
		2-way	.001	0	NA	.001	100	0	NA
		4-way	.001	0	NA	.001	100	0	NA
	32	D.M.	.001	0	NA	.001	100	0	NA
		2-way	.001	0	NA	.001	100	0	NA
		4-way	.001	0	NA	.001	100	0	NA
	64	D.M.	.001	0	NA	0	NA	.001	100
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA

Table B.5. Compulsory, capacity, and conflict misses of Mgrid

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	5.09	0.02	0.45	3.51	69.0	1.56	30.6
		2-way	3.78	0.02	0.6	3.51	92.8	0.25	6.56
		4-way	3.54	0.02	0.6	3.51	99.2	0.01	0.14
	16	D.M.	4.31	0.02	0.53	3.48	80.8	0.81	18.7
		2-way	3.48	0.02	0.66	3.48	99.3	0	NA
		4-way	3.5	0.02	0.65	3.48	99.6	0	NA
	32	D.M.	3.82	0.02	0.6	3.45	90.2	0.35	9.17
		2-way	3.39	0.02	0.67	3.37	99.3	0	NA
		4-way	3.45	0.02	0.66	3.43	99.3	0	NA
	64	D.M.	3.6	0.02	0.63	3.27	91.0	0.3	8.34
		2-way	3.3	0.02	0.69	3.27	99.3	0	NA
		4-way	3.3	0.02	0.69	3.27	99.3	0	NA
Inst Cache	8	D.M.	.005	0	NA	.005	100	0	NA
		2-way	.005	0	NA	.005	100	0	NA
		4-way	.005	0	NA	.005	100	0	NA
	16	D.M.	.005	0	NA	.005	100	0	NA
		2-way	.004	0	NA	.004	100	0	NA
		4-way	.004	0	NA	.004	110	0	NA
	32	D.M.	.001	0	NA	0	NA	.001	100
		2-way	.001	0	NA	0	NA	.001	100
		4-way	.001	0	NA	0	NA	.001	100
	64	D.M.	.001	0	NA	0	NA	.001	100
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA

Table B.6. Compulsory, capacity, and conflict misses of Su2cor

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	12.9	0.04	0.31	11.6	89.7	1.3	10.0
		2-way	11.4	0.04	0.35	11.4	99.7	0	NA
		4-way	11.4	0.04	0.35	11.4	99.7	0	NA
	16	D.M.	12.2	0.04	0.33	11.0	90.1	1.17	9.59
		2-way	11.1	0.04	0.36	11.0	99.0	0.07	0.6
		4-way	11.2	0.04	0.36	11.0	98.8	0.1	0.89
	32	D.M.	11.7	0.04	0.34	10.8	92.2	0.88	7.46
		2-way	10.8	0.04	0.37	10.8	99.8	0	NA
		4-way	10.9	0.04	0.37	10.8	99.6	.004	0.04
	64	D.M.	11.3	0.04	0.35	10.6	93.9	0.65	5.73
		2-way	10.6	0.04	0.38	10.6	99.6	0	NA
		4-way	10.7	0.04	0.38	10.6	99.6	.002	0.02
Inst Cache	8	D.M.	0.46	0	NA	.004	0.86	0.46	99.1
		2-way	0.24	0	NA	.004	1.64	0.24	98.3
		4-way	0.26	0	NA	.004	1.55	0.25	98.4
	16	D.M.	0.32	0	0.03	0	NA	0.32	99.9
		2-way	0.03	0	NA	0	NA	0.03	100
		4-way	0	0	NA	0	NA	0	NA
	32	D.M.	0.06	0	NA	0	NA	0.06	100
		2-way	0.02	0	NA	0	NA	0.02	100
		4-way	0	0	NA	0	NA	0	NA
	64	D.M.	0.04	0	NA	0	NA	0.034	100
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA

Table B.7. Compulsory, capacity, and conflict misses of Swim

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	63.6	0.06	0.09	9.5	14.9	54.0	85.0
		2-way	69.0	0.06	0.08	9.50	13.8	59.5	86.2
		4-way	58.2	0.06	0.1	9.5	16.3	48.6	83.6
	16	D.M.	29.2	0.06	0.2	9.5	32.5	19.7	67.3
		2-way	32.3	0.06	0.18	9.5	29.4	22.7	70.4
		4-way	34.1	0.06	0.17	9.50	27.9	24.5	72.0
	32	D.M.	9.24	0.06	0.62	7.41	80.2	1.77	19.2
		2-way	7.79	0.06	0.74	7.41	95.1	0.32	4.15
		4-way	8.08	0.06	0.71	7.41	91.8	0.61	7.52
	64	D.M.	7.49	0.06	0.77	7.41	98.9	0.02	0.31
		2-way	7.48	0.06	0.77	7.41	99.1	0.01	0.16
		4-way	7.47	0.06	0.77	7.41	99.2	0	NA
Inst Cache	8	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	16	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	32	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	64	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA

Table B.8. Compulsory, capacity, and conflict misses of Tomcatv

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	50.5	0.06	0.12	13.1	25.9	37.4	73.9
		2-way	43.2	0.06	0.14	13.1	30.4	30.0	69.5
		4-way	17.1	0.06	0.35	13.1	76.6	3.9	23.1
	16	D.M.	16.6	0.06	0.36	13.1	78.8	3.48	20.9
		2-way	18.1	0.06	0.34	13.1	72.6	4.9	27.1
		4-way	16.2	0.06	0.37	13.1	80.8	3.06	18.9
	32	D.M.	9.88	0.06	0.61	12.3	124	-2.4	-24
		2-way	9.78	0.06	0.62	12.3	125	-2.5	-25
		4-way	9.78	0.06	0.62	12.3	125	-2.5	-25
	64	D.M.	8.97	0.06	0.68	8.85	98.7	0.06	0.61
		2-way	8.91	0.06	0.68	8.85	99.3	.001	0.01
		4-way	8.91	0.06	0.68	8.85	99.3	.003	0.03
Inst Cache	8	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	16	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	32	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	64	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA

Table B.9. Compulsory, capacity, and conflict misses of Turb3d

	SZ KB	ASSO	MISS %	COM.MISS% ACT.	COM.MISS% REL.	CAP.MISS% ACT.	CAP.MISS% REL.	CON.MISS% ACT.	CON.MISS% REL.
Data Cache	8	D.M.	8.43	0.15	1.8	2.14	25.4	6.14	72.8
		2-way	6.16	0.15	2.5	2.14	34.8	3.86	62.7
		4-way	4.17	0.15	3.6	2.14	51.3	1.88	45.1
	16	D.M.	6.64	0.15	2.3	2.07	31.2	4.4	66.5
		2-way	5.6	0.15	2.7	2.07	36.9	3.39	60.4
		4-way	3.99	0.15	3.8	2.07	51.8	1.77	44.4
	32	D.M.	5.35	0.15	2.8	1.75	32.6	3.5	64.6
		2-way	4.76	0.15	3.2	1.75	36.7	2.86	60.1
		4-way	3.54	0.15	4.3	1.75	49.4	1.64	46.3
	64	D.M.	4.09	0.15	3.7	1.24	30.2	2.70	66.1
		2-way	3.47	0.15	4.4	1.24	35.7	2.08	59.9
		4-way	2.42	0.15	6.2	1.24	51.1	1.03	42.7
Inst. Cache	8	D.M.	0.07	.00002	0.03	.001	1.2	0.07	98.8
		2-way	0.02	.00002	0.1	.001	4.2	0.02	95.7
		4-way	0.01	.00002	0.2	.001	7.3	0.01	92.5
	16	D.M.	0.05	.00002	0.05	.00002	0.05	0.05	99.9
		2-way	.001	.00002	4.0	.00002	4.0	.0005	92.0
		4-way	0	0	NA	0	NA	0	NA
	32	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	64	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA

Table B.10. Compulsory, capacity, and conflict misses of Wave5

	SZ	ASSO	MISS	COM.MISS%		CAP.MISS%		CON.MISS%	
	KB		%	ACT.	REL.	ACT.	REL.	ACT.	REL.
Data Cache	8	D.M.	27.9	0.14	0.50	10.9	39.0	16.9	60.5
		2-way	25.8	0.14	0.54	10.9	42.2	14.8	57.3
		4-way	22.5	0.14	0.62	10.9	48.4	11.5	51.0
	16	D.M.	21.7	0.14	0.64	10.7	49.5	10.8	49.9
		2-way	21.7	0.14	0.64	10.7	49.3	10.9	50.1
		4-way	21.7	0.14	0.64	10.7	49.5	10.8	49.9
	32	D.M.	15.4	0.14	0.91	8.14	52.9	7.11	46.2
		2-way	16.3	0.14	0.86	8.14	49.9	8.04	49.2
		4-way	16.9	0.14	0.83	8.14	48.3	8.58	50.9
	64	D.M.	7.26	0.14	1.92	6.58	90.6	0.54	7.49
		2-way	6.88	0.14	2.03	6.58	95.6	0.16	2.33
		4-way	5.99	0.14	2.33	6.58	109	-0.7	-12
Inst Cache	8	D.M.	0.15	0	NA	.003	2.0	0.15	97.9
		2-way	0.08	0	NA	.003	3.6	0.08	96.4
		4-way	0.07	0	NA	.003	4.49	0.06	95.5
	16	D.M.	0.04	0	NA	0	NA	0.04	100
		2-way	0.02	0	NA	0	NA	0.02	100
		4-way	0	0	NA	0	NA	0	NA
	32	D.M.	0	0	NA	0	NA	0	NA
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA
	64	D.M.	0.01	0	NA	0	NA	0.01	100
		2-way	0	0	NA	0	NA	0	NA
		4-way	0	0	NA	0	NA	0	NA

REFERENCES

[1] S. Adva, and M. Hill, "Weak Ordering - A New Definition," *Proc. of 17th Int'l Symp. on Computer Architecture*, May 1990, pp. 2–14.

[2] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache Performance of Operating Systems and Multiprogramming," *ACM Trans. Computer Systems*, Vol. 6(4), Nov. 1988, pp. 393–431.

[3] A. Agarwal, and S. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches," *Proc. 20th Int'l Symp. on Computer Architecture*, Jun. 1993, pp. 179–190.

[4] C. Anderson, and J.L. Baer, "Two Techniques for Improving Performance on Bus-Based Multiprocessor," *Proc. 1st. High Performance Computer Architecture,* 1995, pp. 264–275.

[5] J. Archibald, and J.-L. Baer, "Cache-Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. on Computer Systems*, Vol. 4, No. 4, Nov. 1986, pp. 273–298.

[6] W. Bolosky, and M. Scott, "False Sharing and its Effect on Shared-Memory Performance," *Proceedings of 4th USENIX Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS),* Sep. 1993, pp. 57–72.

[7] B. Calder, D. Grunwald, and J. Emer, "Predictive Sequential Associative Cache," *Proc. 2nd Symp. on High-Performance Computer Architecture*, San Jose, CA, Jan. 1996, pp. 244–253.

[8] J. Carter, J. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," *Proc. of 13th Symp. on Operating System Principles,* Oct. 1991, pp. 152–164.

[9] J. Carter, J. Bennett, and W. Zwaenepoel, "Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems," *ACM Trans. on Computer Systems,* Vol 13(3), Aug. 1995, pp. 205–243.

[10] J. Chang, H. Chao, and K. So, "Cache Design of A Sub-Micron CMOS System/370," *Proc. 14th Int'l Symp. on Comp. Arch.*, Pittsburgh, PA, Jun. 1987, pp. 208–213.

[11] T.F. Chen, and J.L. Baer, "Reducing Memory Latency via Non-Blocking and Prefetching Caches," *Proc. 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Sep. 1992, pp. 51–61.

[12] W.Y. Chen, S.A. Mahlke, P.P. Chang, and W.M. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," *Proc. 24th Int'l Symp. on Microarchitecture* , 1991.

[13] B. Chung, and J. Peir, "LRU-Based Column Associative Caches," *Comp. Arch. News*, Vol. 26(2), May 1998, pp. 9–17.

[14] M. Dubois, J. Wang, L. Barroso, K. Lee, and Y. Chen, "Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs," *Proc. of the 1991 Conf. on Supercomputing*, Nov. 1991, pp. 197-206.

[15] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom, "The Detection and Elimination of Useless Misses in Multiprocessors," *Proc. of 20th Int'l Symp. on Computer Architecture*, Jun. 1993, pp. 88–97.

[16] S. Eggers, and T. Jeremiassen, "Eliminating False Sharing," *Proc. of 1991 Int'l Conf. on Parallel Processing*, Aug. 1991, pp. I-377–I-381.

[17] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. of 17th Int'l Symp. on Computer Architecture*, May 1990, pp. 15–26.

[18] A. Gonzalez, M. Valero, N. Topham, and J.M. Parcerisa, "Eliminating Cache Conflict Misses Through XOR-Based Placement Functions," *Proc. 11th Int'l Conference Supercomputing*, Vienna, Austria, 1997, pp. 76–83.

[19] E. Gornish, E. Granston, and A. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierachies," *Proc. Int'l Conf. on Supercomputing*, 1990, pp. 354–368.

[20] D. Greenley, J. Bauman, D. Chang, D. Chen, R. Eltejaein, P. Ferolito, P. Fu, R. Garner, D. Greenhill, H. Grewal, K. Holdbrook, B. Kim, L. Kohn, H. Kwan, M. Levitt, G. Maturana, D. Mrazek, C. Narasimhaiah, K. Normoyle, N. Parveen, P. Patel, A. Prabhu, M. Tremblay, M. Wong, L. Yang, K. Yarlagadda, R. Yu, R. Yung, and G. Zyner, "Ultrasparc: The Next Generation Superscalar 64-bit SPARC," *Proc. COMCON'95*, pp. 442–451.

[21] J. Hennessy, and D. Patterson, "Computer Architecture A Quantitative Approach," 2nd edition, *Mogan-Kaufmann*, 1990.

[22] J. Hennessy, and D. Patterson, "Computer Organization and Design," 2nd edition, *Mogan-Kaufmann*, 1996.

[23] M. Hill "A Case for Direct-Mapped Caches," *IEEE Computer*, Vol. 21(12), Dec. 1988, pp. 25–40.

[24] R. Hyde, and B. Fleisch, "An Analysis of Degenerate Sharing and False Coherence," *Journal of Parallel and Distributed Computing*, Vol 34(2), May 1996, pp. 183–195.

[25] L. Iftode, J.P. Singh, and K. Li, "Understanding Application Performance on Shared Virtual Memory Systems," *Proc. of 23rd Int'l Symp. on Computer Architecture*, May 1996, pp. 122–133.

[26] Intel Corporation, "Info @ Intel: The Pentium Pro Processor," Mar. 1996.

[27] T. Johnson, and W. Hwu, "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis," *Proc. 24th Int'l Symp. Comp. Arch.*, Denver, CO, Jun. 1997, pp. 315–326.

[28] D. Joseph, and D. Grunwald, "Prefetching using Markov Predictors," *Proc. 24th Int'l Symp. on Computer Architecture*, Denver, CO, Jun. 1997, pp. 252–263.

[29] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of A Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Int'l Symp. on Computer Architecture*, May 1990, pp. 364–373.

[30] N. Jouppi, and S. Wilton, "Tradeoffs in Two-Level On-Chip Caching," *Proc. 21st Int'l Symp. on Comp. Arch.*, Chicago, IL, Apr. 1994, pp. 34–45.

[31] T. Juan, T. Lang, and J. Navarro, "The Difference-bit Cache," *Proc. 23rd Int'l Symp. Comp. Arch.*, Philadelphia, PA, May 1996, pp. 114–120.

[32] S. Kaxiras, and J.R. Goodman, "The GLOW Cache Coherence Protocol Extension for Widely Shared Data," *Proc. of Int'l Conf. on Supercomputing* , 1996, pp. 35–43.

[33] P. Keleher, A.L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proc. of 19th Int'l Symp. on Computer Architecture* , May 1992, pp. 13–21.

[34] A.C. Klaiber, and H.M. Levy, "An architecture for software-controlled data prefetching," *Proc. 18th Int'l Symp. on Computer Architecture*, 1991, pp. 43–53.

[35] L. Knotothanassis, M. Scott, and R. Bianchini, "Lazy Release Consistency for Hardware-coherent multiprocessor," *Proc. of Supercomputing '95*, 1995, pp. 1705–1736.

[36] G. Kurpanek, K. Chan, J. Zheng, E. DeLano, and W Bryg, "PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface," *COMPCON Digest of Papers*, San Francisco, CA, Feb. 1994, pp. 375–382.

[37] J. Larus, B. Richards, and G. Viswanathan, "LCM: Memory System Support for Parallel Language Implementation," *Proc. of 6th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 208–218.

[38] J. Laudon, and D. Lenoski, "The SGI origin: A ccNUMA Highly Scalable Server," *Proc. of 24th Int'l Symp. on Computer Architecture* , 1997, pp. 241–251.

[39] D. Lenoski, J. Laudon, K, Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, "The Standard Dash Multiprocessor," *Computer, 25(3),* Mar. 1992, pp. 63–79.

[40] D. Levitan, T. Thomas, and P. Tu, "The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor," *Proc. COMCON'95*, pp. 285–291.

[41] L. Liu, "Cache Design with Partial Address Matching," *MICRO'27*, San Jose, CA, Dec. 1994, pp. 128–136.

[42] L. Liu, "History Table for Set Prediction for Accessing a Set-Associative Cache," *United States Patent No. 5,418,922*, May 1995.

[43] T. Mowry, and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, Jun. 1991, pp. 87–106.

[44] T. Mowry, M.S. Lam, and A. Gupta, "Design and Evaluation of a compiler algorithm for prefetching," *Proc. 5th Intl. Conf. on Architectural Support for Programming Language and Operating Systems*, 1992, pp. 62–73.

[45] S. Palacharla, and R. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," *Proc. 16th Int'l Symp. on Computer Architecture*, Chicago, IL, Apr. 1994, pp. 24–33.

[46] J. Peir, W. Hsu, H. Young, and S. Ong, "Improving Cache Performance with Balanced Tag and Data Paths," *Proc. 7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Oct. 1996, pp. 268–278.

[47] J. Rivers, and E. Davidson, "Reducing Conflicts in Direct-Mapped Caches with A Temporality-Based Design," *Proc. 1996 Int'l Conf. Parallel Processing,* Ithaca, NY, Aug. 1996, pp. 151–162.

[48] V. Santhanam, E.H. Gornish, and W. Hsu, "Data Prefetching on the HP PA-8000," *Proc. Int'l Symp. on Computer Architecture,* 1997, pp. 264–273.

[49] A. Seznec, "A Case for Two-Way Skewed-Associative Caches," *Proc. 20th Int'l Symp. on Comp. Arch.*, San Diego, CA, May 1993, pp. 169–178.

[50] A. Seznec, "DASC Cache," *Proc. 1st Symp. High-Performance Comp. Arch.*, Raleigh, NC, Jan. 1995, pp. 134–143.

[51] W. Shi, W. Hu and M. Zhu, "An Innovative Implementation for Directory-based Cache Coherence in Shared Memory Multiprocessor," *Computer Architecture News*, Vol.25(5), 1997, pp. 2-9.

[52] A. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, Vol. 11(12), Dec. 1978, pp. 7-21.

[53] A. Smith, "Cache Memories," *Computing Surveys*, Vol. 14(3), Sep. 1982, pp. 473–530.

[54] K. So, and R. Rechtschaffen, "Cache Operations by MRU Change," *IEEE Trans. Computers*, Vol. 37(6), Jun. 1988, pp. 700–709.

[55] SPEC CPU95 Benchmark Suite, Version 1.10, Aug. 1995.

[56] Sun Microsystems, "Introduction to Shade," Revision C, Apr. 1993.

[57] J. Torrellas, M. Lam, and J. Hennessy, "Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates," *Proc. of 1990 Int'l Conf. on Parallel Processing*, Aug. 1990, pp. II-266–II-270.

[58] TPC Council, "TPC Benchmark C, Standard Specification, Rev. 3.6.2," Jun. 1997.

[59] J.E. Veenstra, "Mint Tutorial and User Manual," *TR 452. Computer Science Department, University of Rochester,* Jul. 1993.

[60] J.E. Veenstra, and R.J. Fowler, "Mint: A Front End for Efficient Simulation of Shared-Memory Multiprocessors," *Proc. of 2nd Int'l Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication System(MASCOTS '94),* Jan. 1994, pp 201–207.

[61] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. of 22nd Int'l Symp. on Computer Architecture*, Jun. 1995, pp. 24–36.

[62] C. Zhang, X. Zhang, and Y. Yan, "Two Fast and High-Associativity Cache Schemes," *IEEE Micro*, Vol. 17(5), Sep./Oct. 1997, pp. 40–49.

BIOGRAPHICAL SKETCH

Yongjoon Lee was born on March 24, 1963 in a small town in southern of Korea. He received a bachelor of science degree from Yonsei University, Seoul, Korea, with a major in electrical engineering. He received the master of science degree from the Stevens Institute of Technology, Hoboken, New Jersey, in May 1992. He will receive a Ph. D. in computer and information science and engineering from the University of Florida in August 1999.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jih-Kwon Peir, Chairman
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Timothy Davis
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Sanguthevar Rajasekaran
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Sartaj Sahni
Distinguished Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Kenneth K. O
Associate Professor of Electrical and
Computer Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

August 1999

M.J. Ohanian
Dean, College of Engineering

Winfred M. Phillips
Dean, Graduate School